2

Compositional Modelling: The Formal Perspective

Hussein Zedan,* Antonio Cau and Ben Moszkowski
Software Technology Research Laboratory
De Montfort University, Leicester
E-mail: {zedan,cau,benm}@dmu.ac.uk

Abstract

We provide a formal framework within which an Information System (IS) could be modelled, analysed, and verified in a compositional manner. Our work is based on Interval Temporal Logic (ITL) and its programming language subset Tempura. This is achieved by considering IS, of an enterprise, as a class of *reactive* systems in which it is continually reacting to asynchronously occurring events within a given period of time. Such a reactive nature permits an enterprise to pursue its business activities to best compete with others in the market place. The technique is illustrated by applying it to a small case study from *Public Service Systems* (PSS).

Keywords : Information System, Modelling, Lean Formal methods, Interval Temporal Logic, Simulation

*The author wishes to acknowledge the funding received from the U.K. Engineering and Physical Sciences Research Council (EPSRC) through the Research Grant GR/M/02583

1 INTRODUCTION

There are many important and challenging issues that arise in modelling Information Systems (IS) within an enterprise. Their complexity is expected to increase as market demands are rapidly changing which ultimately lead to large shifts in the strategic direction of an enterprise. Finance, Public service, work-flow and shop-floor manufacturing systems are only a few examples.

A wide range of production systems pose a diversity of problems all along their life cycle. They are often complex, subtle and multifaceted [34,35]. There is a growing demand, for example in a market-driven manufacturing domain, for methods, languages, tools to model, analyse, build and re-build a manufacturing enterprise which is facing adaptation in an increasingly dynamic situations.

In response, various frameworks were suggested. The most notable approach is the *Soft Systems* framework [5, 32, 33]. Other methods, e.g. OMT [22], Fusion [6] and GRAPES [10], exist that model systems at different levels of abstraction and under different views. Within the process of modelling they provide description techniques like entity/relation-diagrams and their object-oriented extensions, state automata, sequence charts or data-flow diagrams. A critical point of these methods is the lack of precise semantic description. Such a precise underpinning allows proper formal analysis and verification of systems.

The definition of the description techniques as well as the relationships between different description level of a method is usually only given informally. This indeed raises ambiguity and vague interpretation of the semantics of the used modelling concepts. Issues of *consistency* and *completeness* at even a single description level can only be tackled informally. As a consequence CASE-Tools often do not cause the expected gain in productivity: The information which can be acquired by the use of methods is, because of the deficient semantic foundation of the methods, not very evident. As a result, the functionality of most tools is restricted to document editing and managing functions.

What is needed is a disciplined, systematic, compositional and rigorous methodology which is essential for attaining a 'reasonable' level of dependability and trust in these systems. This implies that the modelling of an IS must be treated as an engineering discipline with a proper semantic foundations.

Recently, various approaches to formalise methods for systems were suggested. Well known are the so-called *meta-models*, originating in the context of tool integration (e.g. [9, 28, 30]). Within manufacturing systems, [1, 23] proposed a technique that avoids a *Babel Tower* to rise, where the different people at each stage of the design and operation can seldom communicate. For this they suggested the use of Petri net [21]. In addition, the RAISE method [24, 25] was used to model IS and enterprises [8, 11, 12].

However in some models almost only abstract syntax of the description tech-

niques is captured. In addition, some properties are hard to express in some formalisms whereas they are easy to express in others. Specifications written in a formalism that is not executable nor amenable to algorithmic verification methods can be hard to debug.

In this paper we present a compositional formal framework for modelling general complex systems and its suitability to Information Systems. Our framework enjoys a number of advantages:

- it is *compositional*. This facilitates the modular design and maintenance of complex systems from both sequential and parallel sub-components.
- it is supported by a refinement calculus allowing the systematic derivation of systems through correctness preserving steps;
- it establishes a reliable link between abstract specification, refined design, co-simulation and the ultimate concrete implementation of system.

The presentation of the various aspects of the framework is deliberately kept short due to lack of space. However, we will refer the reader to published work that give full treatment to the work.

The paper is organised as follows. A brief description of Interval Temporal Logic (ITL) is given in Sect. 2. We then introduce in Sect. 3 a case study from the Public Service System, which will be used as a vehicle to explain our compositional modelling theory of Sect. 4. An outline of our tool support is given in Section 5. We conclude with some remarks in Section 6.

2 ITL

We base our work on Interval Temporal Logic (ITL) and its programming language subset Tempura [18]. Our selection of ITL is based on a number of points. It is a flexible notation for both propositional and first-order reasoning about periods of time. Unlike most temporal logics, ITL can handle both sequential and parallel composition and offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and projected time [19]. Timing constraints are expressible and furthermore most imperative programming constructs can be viewed as formulas in a slightly modified version of ITL [3]. Tempura provides an executable framework for developing and experimenting with suitable ITL specifications.

We first give an overview of ITL and the present our compositional theory.

2.1 ITL: Syntax and Semantics

An interval is considered to be a (in)finite sequence of states, where a state is a mapping from variables to their values. The length of an interval is equal to one less

Table 1: Syntax of ITL

Expressions	
$e ::=$	$\mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \imath a{:}\, f$
Formulas	
$f ::=$	$p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathsf{skip} \mid f_1 ; f_2 \mid f^*$

than the number of states in the interval (i.e., a one state interval has length 0).

The syntax of ITL is defined in Table 1 where μ is an integer value, a is a static variable (doesn't change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol, p is a predicate symbol.

The informal semantics of the most interesting constructs are as follows:

- $\imath a{:}\, f$: the value of a such that f holds.
- skip: unit interval (length 1).
- $f_1 ; f_2$: holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval.
- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds.

These constructs enables us to define programming constructs like assignment, if then else, while loop etc. In section A we list some frequently used abbreviations.

2.2 Data Representation in ITL

Introducing type system into specification languages has its advantages and disadvantages. An untyped set theory is simple and is more flexible than any simple typed formalism. Polymorphism, overloading and subtyping can make a type system more powerful but at the cost of increased complexity [15]. While types serve little purpose in hand proofs, they do help with mechanised proofs.

There are two basic inbuilt types in ITL (which can be given pure set-theoretic definitions). These are integers $\mathcal{N}$ (together with standard relations of inequality and quality) and Boolean (*true* and *false*). In addition, the executable subset of ITL (Tempura) has basic types: integer, character, Boolean, list and arrays.

Further types can be built from these by means of $\times$ and the power set operator, $\mathcal{P}$ (in a similar fashion as adopted in the specification language Z).

For example, the following introduces a variable x of type T

$$(\exists x : T) \cdot f \quad \widehat{=} \quad \exists x \cdot type(T) \wedge f$$

Here $type(T)$ denotes a formula describing the desired type. For example, $type(T)$ could be $0 \leq x \leq 7$ and so on. Although this might seem to be rather inexpressive type system, richer type can be added following that of Spivey [29].

3 PUBLIC SERVICE SYSTEMS: A CASE STUDY

In this section we introduce a case study of *PSS* (Public Service Systems) [8] that will illustrate our compositional modelling theory of Sect. 4. A PSS refers to any system, mechanical or otherwise, that either provides some services too a queue of clients or processes some queue of clients in some way. Examples of a PSS include a supermarket, cash points and petrol station. Our chosen example is that of a supermarket.

3.1 System description

A supermarket consists of a buying area, where the customers collect the items they wish to purchase, and one or more cash points, where the customers queue up to pay for those items. Each cash point generally has only one queue of customers, and a customer can join any queue and can move freely between queues (because all cash points generally offer the same service). Customer may only join the rear of a queue.

The basic properties of the supermarket and its cash points and customers are summarised as follows:

- **Location of clients.** A customer may be in any of the following locations:
  1. outside the supermarket
  2. in the buying area
  3. in a queue waiting to pay
  4. at a cash point.
- **Movements of clients.** A customer may move between the following locations:
  1. from outside the supermarket to the buying area. This is not possible if the supermarket is closing.
  2. from the buying area to outside. This is possible if either the required item is not available or the queues are too long!

3. from the buying area to the rear of the queue at some cash point. This is only possible if the cash point is not closing.
4. from the queue to the buying area (this is only possible if the customer is not currently being served.)
5. from the front of the queue to the cash point associated to the queue. This is only possible if the cash point is free and not closed.
6. from any position in the queue to the rear of another queue. This is only possible if the new queue is not closing.
7. from a cash point to outside the supermarket.

- **Status of the cash point**. There are three states:

  1. *closed* (i.e., unable to serve).
  2. *open* (i.e., able to serve customers).
  3. *closing* (i.e., able to serve customers currently in the queue but not new customers).

  A cash point which is not closed may additionally be in either of the following states:

  1. *busy* (currently serving a customer)
  2. *free* (currently not serving a customer).

- **Operation of cash points**.

  1. *open*
  2. *close queue*
  3. *close*
  4. *begin processing* (of a customer)
  5. *end processing* (of a customer)

- **Status of supermarket**

  1. *open*
  2. *closed*
  3. *closing*

- **Operation on supermarket**

  1. *open*
  2. *close entrance*
  3. *close*

4 COMPOSITIONAL MODELLING

Within the area of formal specification and verification, the term *compositionality* is often used to refer to modular techniques for dealing with sequential and concurrent behaviour. In our case study we can identify the following concurrent components:

1. Client i ($0 \leq i < nclients$),
2. Cashpoint j ($0 \leq j < ncashpoint$), and
3. Supermarket

We now proceed with a formal description of the Supermarket component. As seen in Sect. 3 we have the following possible transitions:

- The supermarket opens: Let *Supermarket* be a state variable with the following values

  $Supermarket = sm\_open$: supermarket is open
  $Supermarket = sm\_closed$: supermarket is closed
  $Supermarket = sm\_closing$: supermarket is closing

  The *supermarket opens* state transition can be described in ITL as follows

  $$
  \begin{array}{l}
  supermarket\_open() \mathrel{\widehat{=}} (\\
  \quad \mathsf{skip} \wedge \\
  \quad \mathsf{if}\ Supermarket = sm\_closed\ \mathsf{then}\ (Supermarket := sm\_open) \\
  \qquad\qquad\qquad\qquad \mathsf{else}\ (stable(Supermarket)) \\
)
  \end{array}
  $$

  Note: when the supermarket is *not closed* then the opening transition has no effect, i.e., the state variable *Supermarket* remains unchanged.

- The supermarket is closing.

  $$
  \begin{array}{l}
  supermarket\_closing() \mathrel{\widehat{=}} (\\
  \quad \mathsf{skip} \wedge \\
  \quad \mathsf{if}\ Supermarket = sm\_open\ \mathsf{then}\ (Supermarket := sm\_closing) \\
  \qquad\qquad\qquad\qquad \mathsf{else}\ (stable(Supermarket)) \\
)
  \end{array}
  $$

- The supermarket closes. This transition is a bit more interesting in that it requires that we can only close the supermarket when there are no remaining clients inside the supermarket.

  $$
  \begin{array}{l}
  supermarket\_close() \mathrel{\widehat{=}} (\\
  \quad \mathsf{skip} \wedge \\
  \quad \mathsf{if}\ Supermarket = sm\_closing \wedge in\_supermarket(nclients) = 0\ \mathsf{then}\ (\\
  \qquad Supermarket := sm\_closed \\
  \quad)\ \mathsf{else}\ (stable(Supermarket)) \\
)
  \end{array}
  $$

Note: $in\_supermarket(nclients)$ is a function defined in Sect. B.3.

- The supermarket remains in the current state. This transition which doesn't change the state of the supermarket is taken when the other components make a transition.

$$supermarket\_unchanged() \mathrel{\widehat{=}} (stable(Supermarket))$$

Together with fact that in initial state of the supermarket is closed we can describe the behaviour of supermarket by the following ITL formula:

$$\begin{array}{l} C_{sm} \mathrel{\widehat{=}} \\ Supermarket = sm\_closed \wedge \\ (\ supermarket\_unchanged()^*; supermarket\_open(); \\ \ \ supermarket\_unchanged()^*; supermarket\_closing(); \\ \ \ supermarket\_unchanged()^*; supermarket\_close() \\ )^* \end{array}$$

i.e., we have first a closed phase followed by an open phase then followed by a closing phase and then we again the closing phase. Note: the $supermarket\_unchanged()^*$ takes cares of how long a phase persists.

The state transition diagram of the supermarket component is shown in Fig. 1.

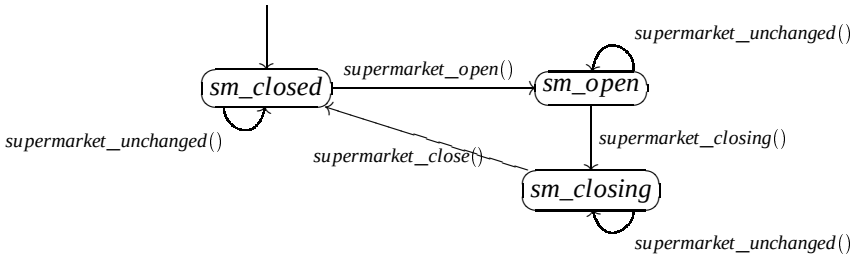


Figure 1: State transition diagram of the supermarket component

The client (C_{c_i}) and cashpoint (C_{p_j}) components can be modelled in a similar way. In appendix B we list the full ITL description of these components.

The complete supermarket system is then describe as follows:

$$Supermarket\_System \mathrel{\widehat{=}} C_{sm} \wedge \bigwedge_i C_{c_i} \wedge \bigwedge_j C_{p_j}$$

5 ANIMATIONS AND EXECUTION

In order to increase accessibility of our technique, it is important that visual supporting tools be developed. This enables the developer to gain more insight of the system and provides a convenient way to rapid prototyping. Towards this goal we have designed and developed an *ITL-Workbench*. In addition to Tempura, the workbench integrates two main components. These are the ITL formal verifier and an animator.

The ITL verifier allows the developer to prove various properties about the model that is being specified. This was done using the Prototype Verification System (PVS [26]). PVS is an interactive environment developed at SRI for writing formal specifications and checking formal proofs. The specification language used in PVS is a strongly typed higher order logic. This specification language is powerful enough to specify the syntax, semantics of ITL, and the proof system of ITL. The powerful interactive theorem prover/proof checker of PVS has a large set of basic deductive steps and the facility to combine these steps into proof strategies. This proof tool was already used for the embedding of the Duration Calculus [27] which is a descendant of ITL. This embedding was a semantical one, an extra external interface was constructed to deal with the syntax of the Duration Calculus. We didn't want to proceed this way because it means an extra interface to be built. Instead we decided to embed ITL semantically and syntactically within PVS. We did this for both finite and infinite ITL. Due to lack of space we have not shown the operation of the verifier. However, we refer the reader to [2].

The animator is a tool that can graphically represent the states generated by Tempura. This gives the developer an easy and fast way to check the developed model. This animator is written in Tcl/Tk [20, 31] using Expect [16]. Each Tempura file can now be accompanied by a Tcl/Tk file which defines the graphics. The Tempura file can issue Tcl/Tk commands to produce the graphics. If one wishes to have graphical output for a particular Tempura program one has to provide a corresponding Tcl/Tk program that produces the graphics. The Tempura tool will control the execution of both programs. And by adding Tk, one can also wrap interactive applications in X11 GUIs.

An example output generated by the Tempura Tool, is shown in Fig. 2– 3. It represents the following example behaviour of the supermarket system.

Time	State
0.	initial state: supermarket and all cashpoints closed, and all clients outside
1.	The supermarket opens
2.	Caspoints 0 and 1 open
3.	Clients 0 and 1 enter
4.	Clients 2 and 3 enter
5.	Clients 0 and 1 queue at cashpoint 0
6.	Client 0 leaves the queue because (s)he forgot something
7.	Client 0 queues at cashpoint 0
8.	Client 2 queues at cashpoint 1
9.	Client 0 switches to cashpoint 1
10.	Cashpoints 0 and 1 begin processing of the first client in their queue
11.	Cashpoints 0 and 1 end processing of respectively Client 1 and 2 who leave the supermarket
12.	Cashpoint 0 is closing
13.	Cashpoint 0 closes
14.	Client 3 leaves the supermarket
15.	Cashpoint 1 begins processing of Client 0
16.	Cashpoint 1 ends processing of Client 0 who leaves the supermarket
17.	Supermarket is closing
18.	Supermarket closes

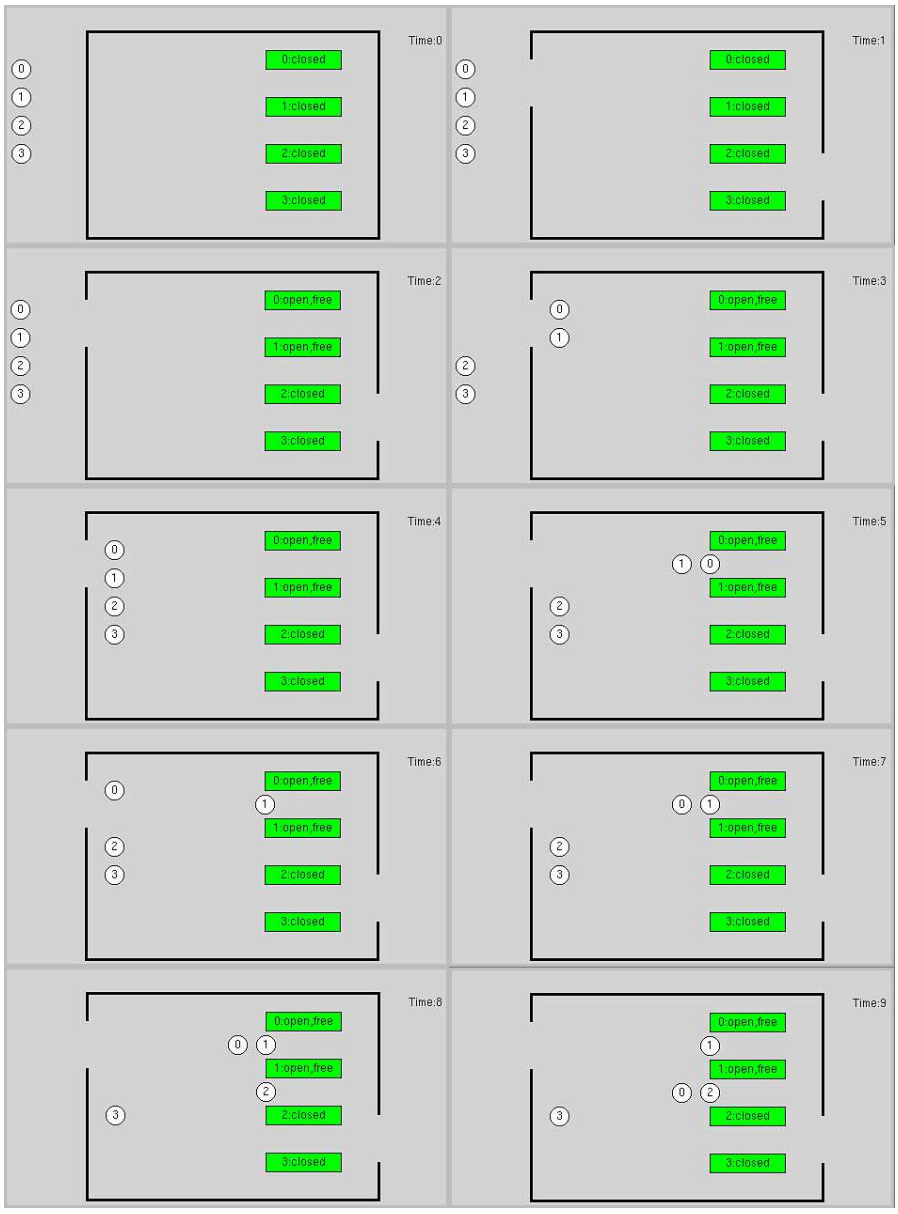


Figure 2: States 0–9

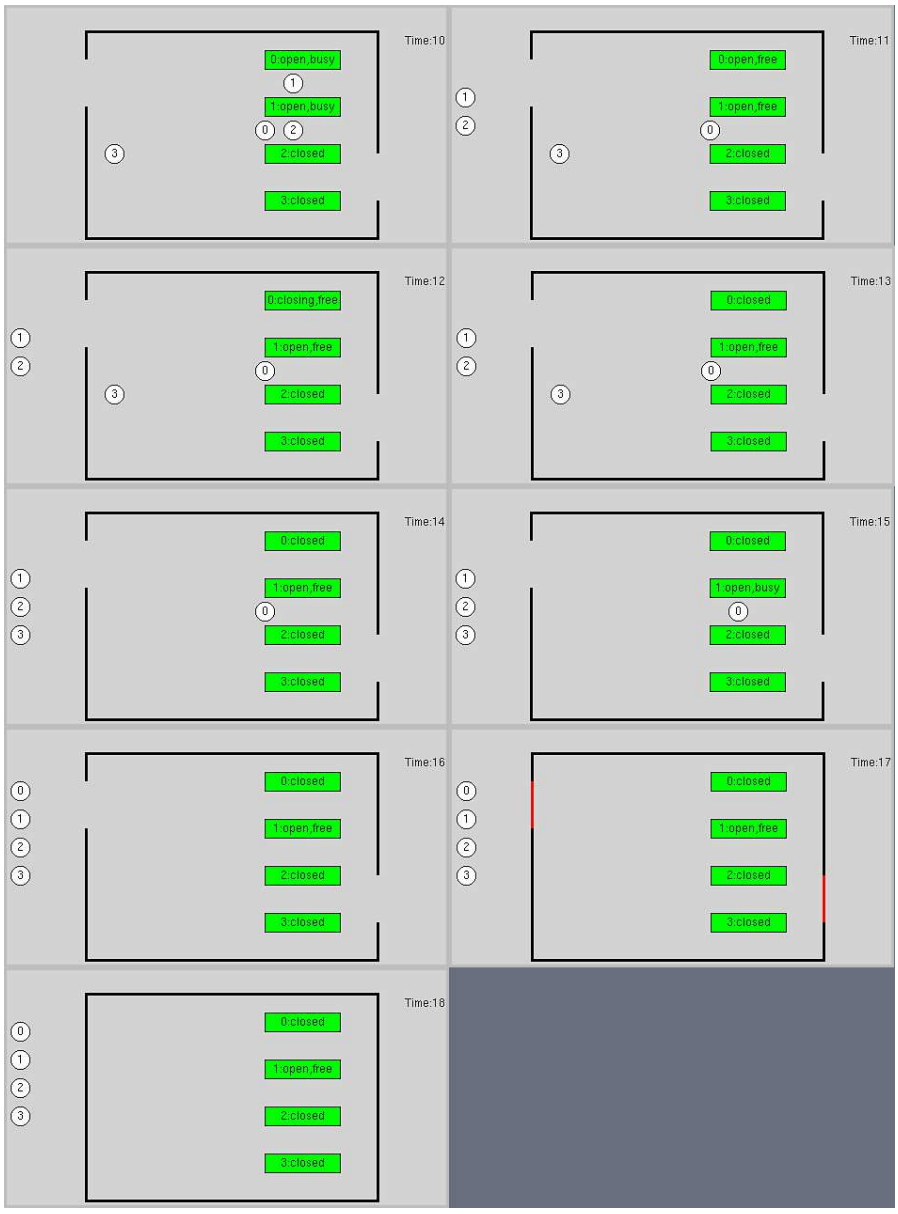


Figure 3: States 9–18

6 DISCUSSION

In order to support system development in an optimal way, description techniques for models of specific system views must be intuitively understandable and be precise enough to ensure an unambiguous and consistent description of the system. In addition, such a technique must be *compositional* allowing the modular description and verification of the system. In this paper, we considered an Information System (IS) to be a reactive system and provided a compositional model for modelling them based on the logical specification formalism known as Interval Temporal Logic (ITL). Moreover, to enhance the acceptance of formal approaches for system modelling and development, integrated tool support must be build. Such tool support should be both graphical and textual that provide a convenient 'user interface' to formal models. To meet this goal, we have interfaced the Tempura tool with Tcl/Tk using Expect. This provides a degree of animation for model construction.

The work presented is only part of a full development technique which

1. establishes a reliable link between abstract ITL specification, refined design, simulation (using Tempura) and its ultimate concrete implementation. The refined design uses a sound refinement calculus that systematically refines an ITL specification into concrete code (see e.g. [3, 4]).

2. provides a framework for the compositional verification of properties about the system.

For example a property of interest may be

$$\vdash\ (\mathit{Supermarket} = \mathit{sm\_closing})\ \supset\ \Diamond(\mathit{in\_supermarket}(\mathit{nclients}) = 0).$$

which states that the Supermarket will eventually be empty once it is closing.

Compositional verification is provided through an *assumptions*/ *commitments*-style framework. The following implication illustrates the use of such style with a system *Sys*:

$$\vdash\ w \wedge \mathit{As} \wedge \mathit{Sys}\ \supset\ \mathit{Co} \wedge \mathit{fin}\, w'.$$

This states that if the state formula w is true in the initial state and the assumption *As* is true over the interval in which *Sys* is operating, then the commitment *Co* is also achieved. Furthermore the state formula w' is true in the interval's final state or is vacuously true if the interval does not terminate.
For example, in our case study, *Sys* may describe the cashpoint Y (i.e., is C_p_Y), *As* is an assumption describing the behaviour of a customer and that *Co* is a temporal formula describing a commitment expected from the cashpoint.

In general, the assumption *As* and the commitment *Co* can be arbitrary ITL formulas. However, when reasoning about a system built out of sequential parts, it

is advantageous to consider certain kinds of assumptions and commitments which readily lend themselves to suitable proof rules.

More specifically, we require that *As* and *Co* be respective fixpoints of the ITL operators $\boxed{a}$ (read "*box-a*") and * (read "*chop-star*") as is now shown:

$$As \equiv \boxed{a} As \ , \qquad Co \equiv Co^* \ .$$

The first equivalence ensures that if the assumption *As* is true on an interval, it is also true in all subintervals. The second ensures that if zero or more sequential instances of the commitment *Co* span an interval, *Co* is also true on the interval itself. The temporal formula $\Box(K = 1)$ (read "*K always equals 1*") is an example of a suitable assumption. The temporal formula $K \leftarrow K$ ("*K's initial and final values on the interval are equal*") is a suitable commitment. Some formulas such as *stable K* ("*K's value remains the same throughout the interval*") can be used both as assumptions and commitments. These are precisely the fixpoints of the ITL operator *keep*, where the formula *keep S*, for some subformula *S*, is true on an interval iff *S* is true on every unit subinterval (i.e., consisting of exactly two adjacent states). For assumptions and commitments obeying the above, the following derivable proof rule is sound:

$$\frac{\begin{array}{l}\vdash \ w \wedge As \wedge Sys \ \supset \ Co \wedge \mathit{fin}\, w' \\ \vdash \ w' \wedge As \wedge Sys' \ \supset \ Co \wedge \mathit{fin}\, w''\end{array}}{\vdash \ w \wedge As \wedge (Sys; Sys') \ \supset \ Co \wedge \mathit{fin}\, w'' \ .} \tag{1}$$

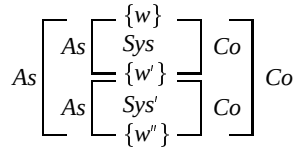
Here is an analogous rule for decomposing a proof for zero or more iterations of a formula *Sys*:

$$\frac{\vdash \ w \wedge As \wedge Sys \ \supset \ Co \wedge \mathit{fin}\, w}{\vdash \ w \wedge As \wedge Sys^* \ \supset \ Co \wedge \mathit{fin}\, w \ .} \tag{2}$$

Similar rules are possible for *if*, *while* and other constructs [1].

Note that our approach only requires assumptions and commitments which are used directly in rules such (1) and (2) to be fixpoints. Compositional proofs about a system in ITL typically also involve reasoning about other kinds of assumptions and commitments as well.

[1] It is advantageous to present proofs in a graphical manner which reflects the structure of a system built from nested sequential parts. The following illustartes this with a proof corresponding to proof rule (1) for basic sequential composition.



We have not given a detail proofs of properties in this paper due to lack of space scope.

Compositional reasoning about *liveness* is also possible. This is primarily achieved by identifying an important class of commitments expressible as ITL formulas of the form $\boxed{m}\diamondsuit S$, where S is itself an arbitrary ITL formula. Let us now informally define the temporal operators $\boxed{m}$ (read "*box-m*") and $\diamondsuit$ (read "*diamond-i*"). A formula $\boxed{m}S$ is true on an interval iff the subformula S is true on all terminal (suffix) subintervals with *more* than one state, that is all the interval's *nonempty* terminal subintervals. Therefore $\boxed{m}$ ignores the last (empty) terminal subinterval consisting of one state and is slightly weaker than the conventional temporal logic operator $\Box$ ("always"). A formula $\diamondsuit S$ is true on an interval iff S is true on some *initial* (prefix) subinterval (which might be the interval itself). Both $\boxed{m}$ and $\diamondsuit$ can be expressed using the basic ITL operator *chop* together with conventional Boolean constructs.

Let w and w' be state formulas. The formula $\boxed{m}(w \supset \diamond w')$ is an instance of a commitment of the form $\boxed{m}\diamondsuit S$ since it can be expressed as $\boxed{m}\diamondsuit(w \supset \diamond w')$. In order to show liveness, we use proof rules such as (1) to modularly establish that a system implies $\boxed{m}(w \supset \diamond w')$ and also (weakly) terminates with $\mathit{fin}(w \supset w')$. These can then be conjoined to obtain the conventional temporal liveness formula $\Box(w \supset \diamond w')$ using the following ITL lemma:

$$\vdash \Box(w \supset \diamond w') \quad \equiv \quad \boxed{m}(w \supset \diamond w') \wedge \mathit{fin}(w \supset w').$$

In addition, we can generalise liveness commitments to be of the form $\boxed{m}(w \supset S; w')$. These subsume those of the form $\boxed{m}\diamondsuit S$ and also facilities reasoning about sequential subcomponents in the commitments themselves. The variant formula $\Box(w \supset S; w')$ can then be used to modularly deal with the equivalence and refinement of specifications by means of derivable proof rules such as the one now given:

$$\vdash \Box(w \supset S; w') \wedge \Box(w' \supset S'; w'') \quad \supset \quad \Box(w \supset S; S'; w'') \ .$$

The compositional approach developed here is not limited to the theoretical analysis of systems. Parts of it can be studied with the Tempura interpreter which can execute useful subsets of ITL. For example, we have been able to empirically test the equivalence of several specifications by synchronously running them in parallel in Tempura. Some of the specifications have annotated assumptions and commitments and others are themselves commitments. In addition, using LITE [14][2], we can verify a number of interesting decidable properties about compositionality.

BIBLIOGRAPHY

[1] ALLA, H, BOBEANU, C.V., AND FILIP, F.G., Modelling and simulation of complex production systems using a Petri net-based formal approach. In *Proc. of 11th European Simulation Multiconference*, 1997.

[2] LITE is a decision procedure for ITL with finite time.

[2] CAU, A., AND MOSZKOWSKI, B. Using PVS for Interval Temporal Logic proofs. Part 1: The Syntactic and semantic encoding. Tech. rep., SERCentre Technical Monograph 14, De Montfort University, 1996.

[3] CAU, A., AND ZEDAN, H, Refining Interval Temporal Logic specifications. In *Transformation-Based Reactive Systems Development*, M. Bertran and T. Rus (Eds.), no. 1231 in LNCS, AMAST, Springer-Verlag, pp. 79–94, 1997.

[4] CAU, A., CZARNECKI, C, AND ZEDAN, H, Designing a Provably Correct Robot Control System using a 'Lean' Formal Method. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, A. P. Ravn and H. Rischel (Eds.), no. 1486 in LNCS, FTRTFT'98, Springer-Verlag, pp. 123–132, 1998.

[5] CHECKLAND, P, Systems Thinking, Systems Practice. *Wiley* (1987).

[6] COLEMAN, D., ET AL., Object-Oriented Development: The Fusion method. Prentice-Hall, Inc. 1994.

[7] FRANCEZ, N, AND PNUELI, A, A proof method for cyclic programs. *Acta Informatica 9*, pp. 133–157, 1978.

[8] GANBAT, T, AND MOORE, R, Specifications of Public Service Systems. UNU/IIST Research Report 129, 1998.

[9] GROSU, R., KLEIN, C., AND RUMPE, B., Enhancing the syslab system model with states. *TUM-I 9631*, Technische Universitat Munchen, 1996.

[10] HELD, G., Sprachbeschreibung GRAPES. SNI AG, Munchen Paderborn, 1990.

[11] JANOWSKI, T., LUGO, G. G., AND HONGJUN, Z., Market-Driven Symbolic Execution of Models of Manufacturing Enterprise. UNU/IIST Research Report 137, 1998.

[12] JANOWSKI, T., LUGO, G. G., AND HONGJUN, Z., Composing Enterprise Models: The Extended and The Virtual Enterprise. UNU/IIST Research Report 131, 1998.

[13] JONES, C. B., Specification and design of (parallel) programs. In *Proceedings of IFIP Congress '83*, Amsterdam, R. E. A. Mason (Ed.), North Holland Publishing Co., pp. 321–332, 1983.

[14] KONO, S., LITE: A little LITL verifier.
Available at *http://rananim.ie.u-ryukyu.ac.jp/kono.*

[15] LAMPORT, L., AND PAULSON, L. C., Should your specification language be typed? SRC Research Report 147, System Research Center, Digital, 1998.

[16] LIBES, D., *Exploring Expect*. O'Reilly & Associates, 1995.

[17] MISRA, J., AND CHANDY, K. M., Proofs of networks of processes. *IEEE Trans. Software Eng. 7*, 4, 417–426, 1981.

[18] MOSZKOWSKI, B., *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, England, 1986.

[19] MOSZKOWSKI, B., Some very compositional temporal properties. In *Programming Concepts, Methods and Calculi*, E.-R. Olderog, Ed., vol. A-56 of *IFIP Transactions*, IFIP, Elsevier Science B.V. (North–Holland), pp. 307–326. Presented at the IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET '94), San Miniato, Italy, 6–10 June, 1994.

[20] OUSTERHOUT, J. K., *Tcl and the Tk Toolkit*. Addison Wesley Publishing Company, Reading, Mass., 1994.

[21] PETRI, C.A., Nets, Time and Space. In; *Theoretical Computer Science, Logics, semantics and Theory of programming*, Vol. 153, pp. 3–48, 1996.

[22] RUMBAUGH, J., ET AL., Object-Oriented Modelling and Design. Prentice-Hall, Inc. (1992).

[23] SILVA, M., AND TERUEL, E., Petri Nets for the design and operation of manufacturing systems. *European Journal of Control*, Vol. 8, pp. 182–199, 1997.

[24] THE RAISE LANGUAGE GROUP, The *RAISE* Specification Language. BCS Practitioners Series, Prentice-Hall, 1992.

[25] THE RAISE LANGUAGE GROUP, The *RAISE* Development Method. BCS Practitioners Series, Prentice-Hall, 1995.

[26] RUSHBY, J. A Tutorial on specification and verification using PVS. In *Proc. of the First Intl. Symp. of Formal Methods Europe FME'93: Industrial-Strength Formal Methods* (Odense, Denmark), J. Woodcock and P. Larsen, Eds., vol. 670 of *LNCS*, Springer-Verlag, pp. 357–406, 1993.

[27] SKAKKEBÆK, J., AND SHANKAR, N., Towards a Duration Calculus proof assistant in PVS. In *Proc. of the 3rd International Symposium Formal Techniques in Real-Time and Fault-Tolerant Systems FTRTFT '94*, H. Langmaack, W.-P. de Roever, and J. Vytopil (Eds.), pp. 660–679, 1994.

[28] D'SOUZA, D., AND WILLS, A., *Catalysis: Practical Rigour and Refinement.* http://www.iconcomp.com

[29] SPIVEY, M., Richer types for Z. Formal Aspects of Computing, 1995.

[30] THOMAS, I., PCTE interfaces: Supporting tools in software engineering environments. *IEEE Software*, pp. 15-23, 1998.

[31] WELCH, B. B., *Practical Programming in Tcl and Tk, 2nd ed.* Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.

[32] WILSON, B., Systems: Concept Methodologies and Applications. *Wiley*, 1984.

[33] WOOD-HARPER, A., ANTILL, L., AND AVISON, D., Information Systems Definition: The Multiview Approach. *Blackwell*, 1985.

[34] ZEIGLER, B.P., Theory of Modelling and simulation. Wiley, NY, 1976.

[35] ZEIGLER, B.P., Multifacitted Modelling and Discrete Event Simulation. Academic Pres, 1984.

A FREQUENTLY USED ITL CONSTRUCTS

In table 2 some frequently used abbreviations are listed.

Table 2: Frequently used abbreviations

$true$	$\widehat{=}$	$0 = 0$	true value
$false$	$\widehat{=}$	$\neg true$	false value
$f_1 \vee f_2$	$\widehat{=}$	$\neg(\neg f_1 \wedge \neg f_2)$	or
$f_1 \supset f_2$	$\widehat{=}$	$\neg f_1 \vee f_2$	implies
$f_1 \equiv f_2$	$\widehat{=}$	$(f_1 \supset f_2) \wedge (f_2 \supset f_1)$	equivalent
$\exists v \bullet f$	$\widehat{=}$	$\neg \forall v \bullet \neg f$	exists
$\bigcirc f$	$\widehat{=}$	$\mathsf{skip}; f$	next
$more$	$\widehat{=}$	$\bigcirc true$	non-empty interval
empty	$\widehat{=}$	$\neg more$	empty interval
inf	$\widehat{=}$	$true; false$	infinite interval
$finite$	$\widehat{=}$	$\neg inf$	finite interval
$\Diamond f$	$\widehat{=}$	$finite; f$	sometimes
$\Box f$	$\widehat{=}$	$\neg \Diamond \neg f$	always
$\circledast f$	$\widehat{=}$	$finite; f; true$	some subinterval
$\boxdot f$	$\widehat{=}$	$\neg(\circledast \neg f)$	all subintervals
$\mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2$	$\widehat{=}$	$(f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$	if then else
$fin\ f$	$\widehat{=}$	$\Box(\mathsf{empty} \supset f)$	final state
$keep\ f$	$\widehat{=}$	$\boxdot(\mathsf{skip} \supset f)$	all unit subintervals
$\bigcirc e$	$\widehat{=}$	$\imath a: \bigcirc(e = a)$	next value
$fin\ e$	$\widehat{=}$	$\imath a: fin\,(e = a)$	end value
$A := e$	$\widehat{=}$	$\bigcirc A = e$	assignment
$e_1 \leftarrow e_2$	$\widehat{=}$	$finite \wedge (fin\ e_1) = e_2$	temporal assignment
$e_1\ gets\ e_2$	$\widehat{=}$	$keep\,(e_1 \leftarrow e_2)$	gets
$stable\ e$	$\widehat{=}$	$e\ gets\ e$	stability
$intlen\,(e)$	$\widehat{=}$	$\exists I \bullet (I = 0) \wedge (I\ gets\ I + 1) \wedge I \leftarrow e)$	interval length e
len	$\widehat{=}$	$\imath a: intlen\,(a)$	interval length

B ITL SPECIFICATION OF THE CLIENT AND CASHPOINT COMPONENTS

B.1 Cashpoint Component

Again we first describe the possible cashpoint transitions for cashpoint Y:
1: Cashpoint Y opens:
Let $Cashpoint[Y]$ be a state variable denoting the state of cashpoint Y with possible values:

$Cashpoint[Y] = cp\_closed$: cashpoint Y is closed
$Cashpoint[Y] = cp\_open\_busy$: cashpoint Y is open and busy
$Cashpoint[Y] = cp\_open\_free$: cashpoint Y is open and free
$Cashpoint[Y] = cp\_closing\_busy$: cashpoint Y is closing and busy
$Cashpoint[Y] = cp\_closing\_free$: cashpoint Y is closing and free

$cashpoint\_open(Y) \mathrel{\widehat{=}} ($
skip $\wedge$
if $Cashpoint[Y] = cp\_closed \vee Cashpoint[Y] = cp\_closing\_free$ then (
$Cashpoint[Y] := cp\_open\_free$
) else (
if $Cashpoint[Y] = cp\_closing\_busy$ then $(Cashpoint[Y] := cp\_open\_busy$
) else $(stable(Cashpoint[Y]))$
)
)

2: Cashpoint Y is closing.

$cashpoint\_closing(Y) \mathrel{\widehat{=}} ($
skip $\wedge$
if $Cashpoint[Y] = cp\_open\_busy$ then $(Cashpoint[Y] := cp\_closing\_busy$
) else (
if $Cashpoint[Y] = cp\_open\_free$ then $(Cashpoint[Y] := cp\_closing\_free$
) else $(stable(Cashpoint[Y]))$
)
)

3: Cashpoint Y closes.

$cashpoint\_close(Y) \mathrel{\widehat{=}} ($
skip $\wedge$
if $Cashpoint[Y] = cp\_closing\_free \wedge queue\_empty(Y)$
then $(Cashpoint[Y] := cp\_closed)$
else $(stable(Cashpoint[Y]))$
)

4: Cashpoint Y remains in the current state.

$cashpoint\_unchanged(Y) \mathrel{\widehat{=}} (stable(Cashpoint[Y]))$

5: Cashpoint Y begins processing the first customer in the queue:

$$
\begin{array}{l}
cp\_begin\_processing(Y) \mathrel{\widehat{=}} (\\
\ \mathsf{skip} \wedge \\
\ \mathsf{if}\ Cashpoint[Y] = cp\_closing\_free\ \mathsf{then}\ (\\
\ \ Cashpoint[Y] := cp\_closing\_busy\\
\ )\ \mathsf{else}\ (\\
\ \ \mathsf{if}\ Cashpoint[Y] = cp\_open\_free\ \mathsf{then}\ (\\
\ \ \ Cashpoint[Y] := cp\_open\_busy\\
\ \ )\ \mathsf{else}\ (stable(Cashpoint[Y]))\\
\ )\\
)
\end{array}
$$

6: Cashpoint Y ends the processing of the customer:

$$
\begin{array}{l}
cp\_end\_processing(Y) \mathrel{\widehat{=}} (\\
\ \mathsf{skip} \wedge \\
\ \mathsf{if}\ Cashpoint[Y] = cp\_closing\_busy\ \mathsf{then}\ (\\
\ \ Cashpoint[Y] := cp\_closing\_free\\
\ )\ \mathsf{else}\ (\\
\ \ \mathsf{if}\ Cashpoint[Y] = cp\_open\_busy\ \mathsf{then}\ (\\
\ \ \ Cashpoint[Y] := cp\_open\_free\\
\ \ )\ \mathsf{else}\ (stable(Cashpoint[Y]))\\
\ )\\
)
\end{array}
$$

The complete specification C_{pY} of cashpoint Y is then

$$
\begin{array}{l}
C_{pY} \mathrel{\widehat{=}}\\
\ Cashpoint[Y] = cp\_closed \wedge\\
\ (cashpoint\_unchanged(Y)^*\,;cashpoint\_open(Y);\\
\ \ (\ (cashpoint\_unchanged(Y)^*\,;cp\_begin\_processing(Y);\\
\ \ \ \ cashpoint\_unchanged(Y)^*\,;cp\_end\_processing(Y))^*;\\
\ \ \ cashpoint\_unchanged(Y)^*\,;cashpoint\_closing(Y);\\
\ \ \ (cashpoint\_unchanged(Y)^*\,;cp\_begin\_processing(Y);\\
\ \ \ \ cashpoint\_unchanged(Y)^*\,;cp\_end\_processing(Y))^*\\
\ \ \ \vee\\
\ \ \ (cashpoint\_unchanged(Y)^*\,;cp\_begin\_processing(Y);\\
\ \ \ \ cashpoint\_unchanged(Y)^*\,;cp\_end\_processing(Y))^*;\\
\ \ \ cashpoint\_unchanged(Y)^*\,;cp\_begin\_processing(Y);\\
\ \ \ cashpoint\_unchanged(Y)^*\,;cashpoint\_closing(Y);\\
\ \ \ cashpoint\_unchanged(Y)^*\,;cp\_end\_processing(Y);\\
\ \ \ (cashpoint\_unchanged(Y)^*\,;cp\_begin\_processing(Y);\\
\ \ \ \ cashpoint\_unchanged(Y)^*\,;cp\_end\_processing(Y))^*\\
\ \ )\,;cashpoint\_unchanged(Y)^*\,;cashpoint\_close(Y)\\
\ )^*
\end{array}
$$

The state transition diagram of cashpoint component Y is shown in Fig. 4.

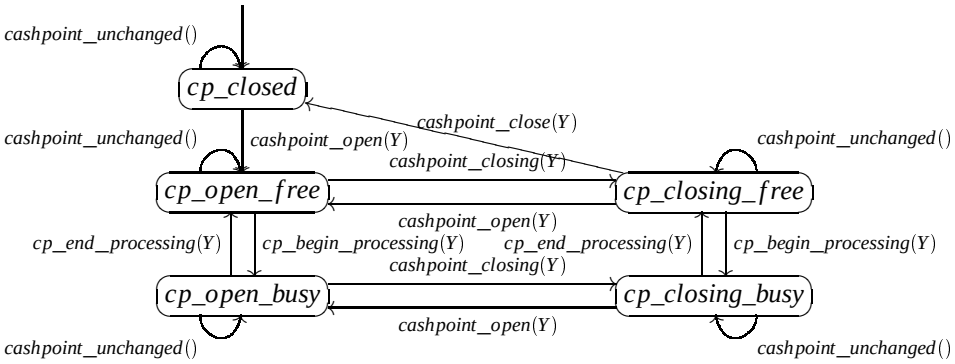


Figure 4: State transition diagram of the cashpoint component

B.2 Client Component

First we specify the possible client transitions for client X:
1: Client X enters the supermarket: Let $Client[X]$ be the state variable describing the state of client X with the following values

$Client[X] = outside$	:	client X is outside the supermarket
$Client[X] = buying$	:	client X is in the buying area
$Client[X] = queuing$	:	client X is in a queue waiting to pay
$Client[X] = served$	:	client X is served at a cashpoint

The transition is then described as follows

$$
\begin{array}{l}
client\_enter(X) \mathrel{\widehat{=}} (\\
\ \mathsf{skip} \wedge stable(Qtime[X]) \wedge \\
\ \mathsf{if}\ Supermarket = sm\_open\ \mathsf{then}\ (\\
\ \ \mathsf{if}\ Client[X] = outside\ \mathsf{then}\ (Client[X] := buying \\
\ \ )\ \mathsf{else}\ (stable(Client[X])) \\
\ )\ \mathsf{else}\ (stable(Client[X])) \\
)
\end{array}
$$

2: Client X leaves the supermarket without paying.

$$
\begin{array}{l}
client\_leave(X) \mathrel{\widehat{=}} (\\
\ \mathsf{skip} \wedge stable(Qtime[X]) \wedge \\
\ \mathsf{if}\ Client[X] = buying\ \mathsf{then}\ (Client[X] := outside \\
\ )\ \mathsf{else}\ (stable(Client[X])) \\
)
\end{array}
$$

3: Client X queues at cashpoint Y:
Let $Qtime[X][Y]$ denote the arrival time of client X at the queue of cashpoint Y (*notthere* if client X is not in the queue of cashpoint Y).
Let furthermore $Timer$ denote the current time.

$$
\begin{array}{l}
client\_queuing(X,Y) \mathrel{\widehat{=}} (\\
\ \mathsf{skip} \wedge \\
\ \mathsf{if}\ Client[X] = buying \wedge (Cashpoint[Y] = cp\_open\_busy \vee \\
\qquad\qquad\qquad\qquad Cashpoint[Y] = cp\_open\_free)\ \mathsf{then}\ (\\
\ Client[X] := queuing \wedge \\
\ (\forall j < ncashpoints \bullet \mathsf{if}\ j = Y\ \mathsf{then}\ Qtime[X][j] := Timer \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ stable(Qtime[X][j])) \\
\ )\ \mathsf{else}\ (stable(Client[X]) \wedge stable(Qtime[X])) \\
)
\end{array}
$$

4: Client X returns to the buying area.

$$
\begin{array}{l}
client\_forgot(X) \mathrel{\widehat{=}} (\\
\ \mathsf{skip} \wedge \\
\ \mathsf{if}\ Client[X] = queuing\ \mathsf{then}\ (\\
\ Client[X] := buying \wedge \\
\ (\forall j < ncashpoints \bullet \mathsf{if}\ j = in\_queue(X)\ \mathsf{then}\ Qtime[X][j] := notthere \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ stable(Qtime[X][j])) \\
\ )\ \mathsf{else}\ (stable(Client[X]) \wedge stable(Qtime[X])) \\
)
\end{array}
$$

5: Client X switches to the queue of cashpoint Z.

$$
\begin{array}{l}
client\_impatient(X,Z) \mathrel{\widehat{=}} (\\
\ \mathsf{skip} \wedge \\
\ \mathsf{if}\ Client[X] = queuing \wedge (Cashpoint[Z] = cp\_open\_free \vee \\
\qquad\qquad\qquad\qquad Cashpoint[Z] = cp\_open\_busy)\ \mathsf{then}\ (\\
\ Client[X] := queuing \wedge \\
\ (\forall j < ncashpoints \bullet \mathsf{if}\ j = in\_queue(X)\ \mathsf{then}\ Qtime[X][j] := notthere \\
\qquad\qquad\qquad\quad \mathsf{else\ if}\ j = Z\ \mathsf{then}\ Qtime[X][j] := Timer \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ stable(Qtime[X][j])) \\
\ )\ \mathsf{else}\ (stable(Client[X]) \wedge stable(Qtime[X])) \\
)
\end{array}
$$

6: Client X is being served:

$$
\begin{array}{l}
client\_served(X) \mathrel{\widehat{=}} (\\
\ \mathsf{skip} \wedge \\
\ \mathsf{if}\ Client[X] = queuing \wedge \\
\quad cl\_served(in\_queue(X)) = X \wedge \\
\quad ((Cashpoint[in\_queue(X)] = cp\_open\_free \wedge \\
\qquad \bigcirc Cashpoint[in\_queue(X)] = cp\_open\_busy) \vee \\
\qquad (Cashpoint[in\_queue(X)] = cp\_closing\_free \wedge \\
\qquad\ \bigcirc Cashpoint[in\_queue(X)] = cp\_closing\_busy)) \\
\ \mathsf{then}\ (Client[X] := served \wedge stable(Qtime[X])) \\
\ \mathsf{else}\ (stable(Client[X]) \wedge stable(Qtime[X])) \\
)
\end{array}
$$

7: Client X pays and leaves the supermarket:

$$
\begin{array}{l}
client\_paid(X) \mathrel{\widehat{=}} (\\
\ \mathsf{skip} \wedge \\
\ \mathsf{if}\ Client[X] = served \wedge \\
\quad ((Cashpoint[in\_queue(X)] = cp\_closing\_busy \wedge \\
\qquad \bigcirc Cashpoint[in\_queue(X)] = cp\_closing\_free) \vee \\
\qquad (Cashpoint[in\_queue(X)] = cp\_open\_busy \wedge \\
\qquad\ \bigcirc Cashpoint[in\_queue(X)] = cp\_open\_free)) \\
\ \mathsf{then}\ (\\
\ Client[X] := outside \wedge \\
\ (\forall j < ncashpoints \bullet \mathsf{if}\ j = in\_queue(X)\ \mathsf{then}\ Qtime[X][j] := notthere \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ stable(Qtime[X][j])) \\
\ )\ \mathsf{else}\ (stable(Client[X]) \wedge stable(Qtime[X])) \\
)
\end{array}
$$

8: Client X remains in the current state.

$$client\_unchanged(X) \mathrel{\widehat{=}} (\mathsf{skip} \wedge stable(Client[X]) \wedge stable(Qtime[X]))$$

The complete specification C_{cX} of Client X is not given but the state transition diagram of client component X is given in Fig. 5.

B.3 Specification of auxiliarly Functions

The following are the functions that are used in the description of the transitions of the various components, these are given to make the specification complete.
A: Determines if queue Y is empty.

$$
\begin{array}{l}
queue\_empty(Y) \mathrel{\widehat{=}} (\\
\ (\forall i < nclients \bullet \mathsf{if}\ Client[i] = queuing \wedge in\_queue(i) = Y \\
\qquad\qquad\qquad\qquad \mathsf{then}\ Qtime[i][Y] = notthere) \\
)
\end{array}
$$

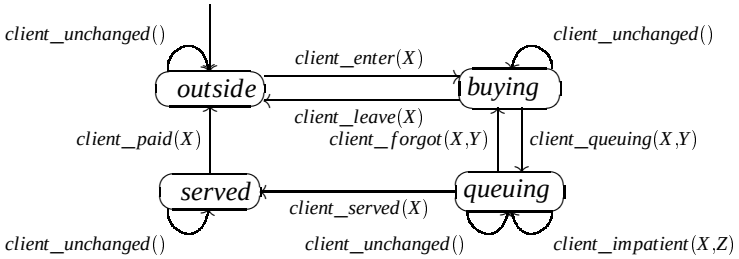


Figure 5: State transition diagram of the client component

B: Find the first customer in queue Y recursively.

$$
\begin{array}{l}
find\_served(n,Y) \mathrel{\widehat{=}} (\\
\mathsf{if}\ n = 0\ \mathsf{then}\ (false)\\
\mathsf{else}\ (\mathsf{if}\ in\_queue(n-1) = Y \wedge Client[n-1] = queuing \wedge\\
\qquad (\forall i < nclients \bullet \mathsf{if}\ in\_queue(i) = Y \wedge Client[i] = queuing \wedge i \neq n-1\ \mathsf{then}\ (\\
\qquad\qquad Qtime[n-1][Y] < Qtime[i][Y] \vee\\
\qquad\qquad (Qtime[n-1][Y] = Qtime[i][Y] \wedge n-1 < i)\\
\qquad\qquad)\\
\qquad)\\
\qquad \mathsf{then}\ (n-1)\ \mathsf{else}\ find\_served(n-1,Y)\\
)\\
)
\end{array}
$$

C: Find the first customer in queue Y.

$$cl\_served(Y) \mathrel{\widehat{=}} (find\_served(nclients,Y))$$

D: Find the customer that has been served at cashpoint Y recursively and is about to leave the supermarket.

$$
\begin{array}{l}
find\_leave(n,Y) \mathrel{\widehat{=}} (\\
\mathsf{if}\ n = 0\ \mathsf{then}\ (false)\\
\mathsf{else}\ (\mathsf{if}\ Client[n-1] = served \wedge in\_queue(n-1) = Y\ \mathsf{then}\ (n-1)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ find\_leave(n-1,Y))\\
)
\end{array}
$$

E: Find the customer that has been served at cashpoint Y and is about to leave the supermarket.

$$cl\_leave(Y) \mathrel{\widehat{=}} (find\_leave(nclients,Y))$$

F: Find out in which queue client X currently is recursively.

$$
\begin{array}{l}
find\_queue(n, X) \mathrel{\widehat{=}} (\\
\ \mathsf{if}\ n = 0\ \mathsf{then}\ (false) \\
\ \mathsf{else}\ (\mathsf{if}\ Qtime[X][n-1] \neq notthere\ \mathsf{then}\ n-1\ \mathsf{else}\ find\_queue(n-1, X)) \\
)
\end{array}
$$

G: Find out in which queue client X currently is.

$$
\begin{array}{l}
in\_queue(X) \mathrel{\widehat{=}} (\\
\ \mathsf{if}\ (\forall i < ncashpoints \bullet Qtime[X][i] = notthere)\ \mathsf{then}\ notthere \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ find\_queue(ncashpoints, X) \\
)
\end{array}
$$

H: Determine the number of clients inside the supermarket recursively.

$$
\begin{array}{l}
in\_supermarket(n) \mathrel{\widehat{=}} (\\
\ \mathsf{if}\ n = 0\ \mathsf{then}\ 0 \\
\ \mathsf{else}\ in\_supermarket(n-1) + (\mathsf{if}\ Client[n-1] \neq outside\ \mathsf{then}\ 1\ \mathsf{else}\ 0) \\
)
\end{array}
$$