A Formal Model for SDL Specifications based on Timed Rewriting Logic

L. J. Steggles[1] and P. Kosiuczenko[2].

[1]Department of Computer Science, University of Newcastle.

email: L.J.Steggles@newcastle.ac.uk

[2]Institut für Informatik, Ludwig–Maximilians–Universität of München.

email: kosiucze@informatik.uni-muenchen.de

Abstract

SDL (Specification and Description Language) is a standard industrial formal description technique for real–time distributed systems which is based on communicating finite state machines. Despite its wide spread use and industrial importance SDL lacks at present a complete and integrated formal semantics. In this paper we begin to develop such a formal semantics for SDL using a new algebraic formalism called Timed Rewriting Logic (TRL). TRL is a specification formalism which extends standard algebraic specification techniques by allowing the dynamic behaviour of systems to be axiomatised using term rewriting rules. The rewrite rules can be labelled with time constraints which provide a means of reasoning about time elapse in real–time systems. The formal semantics we develop captures in an intuitive way the hierarchical structure of SDL specifications and integrates within one formalism the static and dynamic aspects of an SDL system. It also provides a natural basis for analysing, verifying, testing and composing SDL systems. We demonstrate the approach we develop by considering modelling an SDL specification for the so called bump game.

Keywords: SDL, formal description techniques, real–time, rewriting logic, algebraic semantics.

1 Introduction.

SDL (Specification and Description Language) is an industrial standard formal description technique (FDT) for real–time distributed systems. It was developed in the early 1970's by the CCITT (renamed to the ITU-T) as a standard language for the description of telecommunication systems. Since then it has become increasingly popular as an FDT for industrial real–time systems, due impart to it having both a graphical and textual syntax. Various versions of the language have appeared over the years the most recent being an object–oriented version referred to as SDL–92 (see CCITT [1992] and Faergemand and Olsen [1994]). However, in this paper we consider SDL–88 (see CCITT [1988]) an earlier version of the language which is widely used and supported. Despite being described as a formal language SDL does not at present have a complete, integrated and usable formal semantics (in our opinion Annex F of Z.100, CCITT [1988], falls far short of this mark). Given the wide spread industrial use and importance of SDL there is a real need for a natural formal semantics of SDL to be developed which can be used to analyse and verify system specifications. To this end there have been several attempts in the past few years including semantics based on: process algebra (Bergstra and Middelburg [1995]), temporal logic (Leue [1995]) and duration calculus (Mork et al [1996]), stream processing functions (Broy [1991], Holz and Stolen [1995] and Hinkel [1997]). With the exception of Hinkel [1997] these semantics focus on different views of SDL and formalize different parts of the language. In our opinion SDL still lacks a formal semantics which integrates within one unifying framework all the main aspects of the language.

In this paper we begin to address this issue by proposing a new integrated formal operational semantics for SDL using an algebraic formalism called *Rewriting Logic* (RL). RL (see for example Meseguer [1992]) is an extension of standard algebraic specification techniques which allows the dynamic behaviour of systems to be modelled using rewrite rules. In RL the idea is to define the static and functional aspects of a system using standard algebraic specifications and to then view terms over this specification as system states. The dynamic behaviour of the system is then axiomatised by rewrite rules which define the possible concurrent state transitions of the system. RL has been used to define an object–oriented specification language called Maude which is described in Meseguer [1993] and Meseguer and Winkler [1992]. In order to cope with modelling real–time properties a variant of RL called *Timed Rewriting Logic* (TRL) has recently been proposed (see Kosiuczenko and Wirsing [1997]). TRL allows timing constraints in the form of time stamps to be added to rewrite rules enabling us to reason about time elapse in dynamic real–time systems. A timed version of Maude has also been developed based on TRL and we will make extensive use of this object–oriented specification language in the sequel.

In the following we describe in detail how given an SDL specification we can derive a Timed Maude specification which provides a formal model of the original SDL specification. We begin by considering basic SDL specifications (see Belina et al [1991]) and by discussing how these can be modelled in RL. We then introduce time and in particular, timers which we model using TRL. We demonstrate our approach by constructing a TRL specification for an SDL specification of the so called bump game. The semantic model we propose utilises Timed Maudes object–oriented features and uses distinct objects to represent the processes, blocks and channels contained within an SDL system. Thus the structure of our semantics corresponds in a very natural way to that of an SDL specification. We also take advantage of Timed Maudes modular structuring mechanisms and each block in an SDL specification results in a corresponding module specification which imports the necessary subblock or process modules. Thus the resulting operational semantics captures in an intuitive way the hierarchical structure of an SDL specification.

The new semantics we present has a number of key advantages over its predecessors. It incorporates together the different views of an SDL specification including abstract data types, process, block and real-time descriptions. It also provides an intuitive and natural formal basis for analyzing SDL specifications and has the added advantage of efficient tool support. Abstract data types (ADTs) form an integral part of the static description of an SDL system and are based on the algebraic approach of initial algebra semantics (c.f. Annex C and I to Z.100, CCITT [1988]). Since TRL is an algebraic language our semantics allows for the straightforward incorporation of SDL ADT specifications and thus unifies the static and dynamic semantic parts of an SDL specification within one formalism. This means that the unification of different formalisms is an unnecessary task, an area which has consumed a considerable amount of effort in recent years. In our opinion the unifying approach and the use of a formal object-oriented specification language along with efficient tool support makes the new semantics interesting. Other points to note are the similarity of hierarchic block structures with term structures, and that communication only occurs between blocks on the same level. This makes our semantics compositional and allows us to use the term structure in a very natural way.

The paper is organised as follows. In Section 2 we introduce the necessary background material on TRL and Timed Maude. Then in Section 3 we present a brief overview of SDL. We consider modelling SDL specifications in Section 4 and demonstrate our ideas with a simple example of an SDL specification for the so called bump game in Section 5. Finally, in Section 6 we make some concluding remarks.

2 Timed Rewriting Logic and Timed Maude.

In this section we briefly introduce TRL and the object–oriented specification language Timed Maude. For a detailed account of TRL and Timed Maude we refer the interested reader to Kosiuczenko and Wirsing [1995, 1997], while for an example of its use see Olveczky et al [1996].

In the following we assume the reader is familiar with the basic theory of algebraic specification methods (see for example Ehrig and Mahr [1985] and Wirsing [1990]).

2.1 Timed Rewriting Logic.

Rewriting Logic (RL) is an extension of standard algebraic specification techniques which is able to model dynamic system behaviour. In RL the functional and static properties of a system are described by standard algebraic specifications, whereas the dynamic behaviour of the system is modelled using rewrite rules. Terms over a given signature Σ represent the global states (or configurations) of a system and rewrite rules model the dynamic transitions between these states. For a detailed introduction to RL see Meseguer [1992].

Timed Rewriting Logic (TRL) extends RL by allowing timing constraints to be added to rewrite rules. Every time dependent rewrite step in the system is labelled with a time stamp and this allows us to reason about time elapse in real–time systems. In TRL time is modelled abstractly by an archimedean monoid: a partially ordered, directed monoid with the least element 0 which satisfies the archimedean property, i.e. for any x different from 0 and any y, there is an n such that $n * x > y$ (see Kosiuczenko and Wirsing [1997]). In particular, time can be modelled by the natural or real numbers. A timed rewrite rule is a literal written as $t1 - \; r \rightarrow t2$, where $r \in \mathbf{R}^+$ and $t1, t2 \in T(\Sigma, X)_s$ are Σ terms of the same sort s. Informally, this means that $t1$ evolves to $t2$ in time r ($\mathbf{R}^+$ is the domain of the underlying archimedean monoid). The basic rules of the rewriting calculus (see for example Meseguer [1992]) are extended with time labels as follows: transitivity yields the addition of the time elapses; the congruence and replacement rules are modelled by synchronous composition (which allows us to enforce uniform time elapse in all components of a system); and reflexivity is modelled using a 0-time reflexivity rule which allows actions to be interleaved.

(i) **Timed Transitivity (TT).** For each $t_1, t_2, t_3 \in T(\Sigma, X)$ and $r_1, r_2 \in \mathbf{R}^+$ we have the rule

$$\frac{t_1 - \ r_1 \to t_2, \quad t_2 - \ r_2 \to t_3}{t_1 - \ r_1 + r_2 \to t_3}$$

(ii) **Synchronous Replacement (SR).** Let $t, u \in T(\Sigma, X)$, let $\{x_1, \dots, x_n\} = FV(t) \cup FV(u)$ and let $\{x_{i1}, \dots, x_{ik}\} = FV(t) \cap FV(u)$ be the intersection of the free variables of t and u. For each $t_1, \dots, t_n, u_1, \dots, u_n \in T(\Sigma, X)$ and $r \in \mathbf{R}^+$ we have the rule

$$\frac{t - \ r \to u, \ t_{i1} - \ r \to u_{i1}, \dots, t_{ik} - \ r \to u_{ik}}{t(t_1, \dots, t_n) - \ r \to u(u_1, \dots, u_n)}$$

(iii) **Timed Compatibility with = (TC).** For each $t_1, t_2, u_1, u_2 \in T(\Sigma, X)$ and $r_1, r_2 \in \mathbf{R}^+$ we have the rule

$$\frac{t_1 = u_1, \ r_1 = r_2, \ u_1 - \ r_1 \to u_2, u_2 = t_2}{t_1 - \ r_2 \to t_2}$$

(iv) **0-Time Reflexivity (0R).** For each $t \in T(\Sigma, X)$ we have the rule

$$\frac{}{t - \ 0 \to t}$$

Note that synchronous composition combined with irreflexivity implies that no component of a process can stay idle.

2.2 Timed Maude.

Timed Maude is an object-oriented real-time specification language which is based on TRL. Timed Maude extends the language Maude (see Meseguer and Winkler [1992] and Meseguer [1993]) by replacing concurrent rewriting with TRL. An *object* in Maude is represented by a tuple - more precisely by a term - comprising a unique object identifier, the class to which the object belongs and a set of attributes (local state). For example, the term $< p : P \mid state : S, \ saved : n >$ represents an object with object identifier p belonging to the class P. The attribute *state* has value S and the attribute *saved* has value n. A *message* is a term that consists of the message's name, the identifier of the object the message is addressed to and, possibly, parameters (in mixfix notation). A Maude specification or program makes computational progress by rewriting its global state, referred to as its *configuration*. A configuration is a multiset, or bag, of objects and messages. The sorts *Msg* of messages and *Obj* of objects are considered as subsorts of the sort *Conf* of configurations. Formally, a configuration is a term of the form $m_1 \otimes \cdots \otimes m_k \otimes o_1 \otimes \cdots \otimes o_l$, where $\otimes : Conf \times Conf \to Conf$ is the function symbol for multiset union modelling composition, $m_1, \dots, m_k$ are messages (terms of sort *Msg*), and $o_1, \dots, o_l$ are objects (terms of sort *Obj*). Multiset union is commutative, associative and has an identity *null* as the following axioms formalise:

$$x \otimes y = y \otimes x, \quad x \otimes (y \otimes z) = (x \otimes y) \otimes z, \quad x \otimes null = x.$$

For brevity we often omit the symbol $\otimes$ in configurations (i.e. we write $m_1 \dots m_k\ o_1 \dots o_l$ for $m_1 \otimes \cdots \otimes m_k \otimes o_1 \otimes \cdots \otimes o_l$).

The configurations evolve by consuming/producing messages and removing/creating objects. This evolution is specified using timed rewrite rules which allow the elements of a configuration to change in a dynamic way (see for example Kosiuczenko and Wirsing [1997]). The timed rules have the following general form

$$
\begin{array}{l}
m_1 \dots m_n < O_1 : C_1 \mid atts_1 > \dots < O_q : C_q \mid atts_q > - \ r \rightarrow \\
< O_{i1} : C'_{i1} \mid atts'_{i1} > \dots < O_{ik} : C'_{ik} \mid atts'_{ik} > \\
\quad < Q_1 : D_1 \mid atts''_1 > \dots < Q_p : D_p \mid atts''_p > m'_1 \dots m'_l \\
\quad if\ Cond
\end{array}
$$

where $n, q, k, p, l \geq 0$, $r \in \mathbf{R}^+$ and $Cond$ is an optional rule condition or guard. In the above rule the messages $m_1, \dots, m_n$ are consumed, the state and class of objects $O_{i1}, \dots, O_{ik}$ may change while the other original objects are removed, and the new objects $Q_1, \dots, Q_p$ and new messages $m'_1, \dots, m'_l$ are created. The rule takes r time units to be performed. If $q = 1$ (i.e. only one object on left hand side of rule) then the rule is referred to as an *asynchronous rule*, otherwise the rule is referred to as a *synchronous rule*. Note that the above rule allows the components of a configuration to change dynamically (see Kosiuczenko and Wirsing [1997]). We will follow the Maude convention that those attributes of an object not actively involved in a transition (i.e. a rewrite step) maybe omitted. We usually assume an axiom specifying that time may elapse in a system to allow individual components to execute their actions and ensure (via the SR rule of TRL) uniform time elapse in all components, e.g.

$$c_1 \otimes c_2 - \ r \rightarrow c_1 \otimes c_2,$$

for all $r \in \mathbf{R}^+$ and where c_1, c_2 are variables of sort $Conf$.

Timed Maude treats inheritance in the same way as Maude by using subsorting and import lists (see Meseguer [1993]). For example, we can define natural numbers as a subsort of the real numbers (subsorts $Nat < Real$). This notion of subsort can be understood as inclusion which implies that the subsort is not modified. All sorts are assumed to be static unless stated otherwise. For all terms t of a static sort we have the scheme of axioms $t - \ r \rightarrow t$, for $r \in \mathbf{R}^+$, meaning that t does not change in time (although by the rules of TRL its arguments may as in the rule above).

3 Introduction to SDL.

SDL (Specification and Description Language) is a formal description technique (FDT) for real–time distributed systems. It was developed in the early 1970's by the CCITT (now referred to as the ITU-T) as a standard language for the description of telecommunication systems. Since then several versions of the language have evolved, the most recent being an object–oriented version called SDL–92 (see for example Faergemand and Olsen [1994]). This paper is based on SDL–88 (see CCITT [1988]), an earlier version of the language which is widely used and supported at the present time. For an introduction to SDL–88 we recommend Belina and Hogrefe [1989] and Belina et al [1991].

SDL is an FDT based on modelling systems as a collection of communicating finite state machines. It provides both a graphical (SDL/GR) and textual (SDL/PR) syntax, and specifications have a hierarchical structure. An SDL specification describes a *system* and everything external to that system is referred to as the *environment*. We assume that the environment behaves in an SDL like manner. A system consists of a number of *blocks* which communicate with each other and the environment via a number of *channels*. Each block itself then consists of a number of communicating sub–blocks until, at the lowest level, we have what we refer to as the *atomic blocks*. Atomic blocks consist of processes which communicate with each other and the associated block channels via *signal routes*. Note that a block never contains both processes and sub–blocks. This system structure is illustrated in figure 1 where the squares represent blocks and the rounded boxes represent processes. The behaviour of the entire system is derived by combining the behaviour of all the processes in the system. Processes communicate with each other by sending and

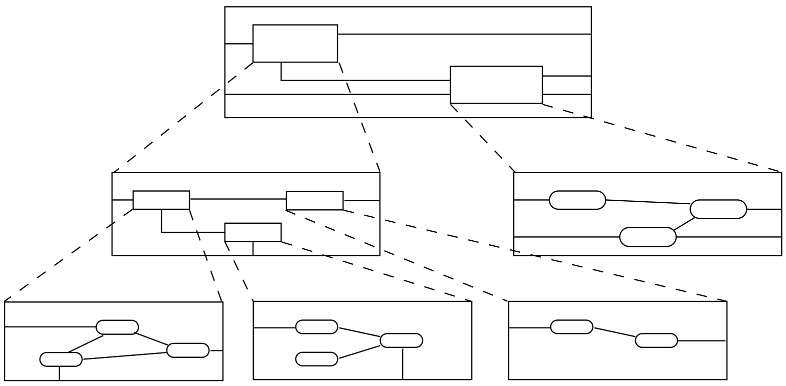

Figure 1: The structure of an SDL system specification.

receiving *signals*. Each process has a unique process identification number (pid) which is normally used to address signals, referred to as *explicit addressing*. However, if the destination of a signal is uniquely defined by the system structure (i.e. only one process can possibly receive the signal) the specification of an explicit address is not required allowing so called *implicit addressing*. Signals sent via signal routes suffer no delay, where as signals sent along channels are assumed to suffer a non–deterministic delay. If two processes are in the same block then communication uses only signal routes. However, for two processes in two different blocks to communicate they must use a combination of signal routes and channels.

A process can be viewed as an extended finite state machine which works autonomously but concurrently with other processes. Each process has a state *start* which is the initial state for the process. Processes receive input signals which can initiate various state transitions according to the processes current state. A process can contain local variables and this forms an implicit state which can be used to influence state transitions. Each process has a single unbounded input queue and all incoming signals from its associated signal routes are placed on this queue in the order they arrive (simultaneous signals are ordered non–deterministically). States are assumed to be stable positions and state transitions are normally triggered by the consumption of a signal from the processes input queue. The consumption of different input signals lead to different states. If the next input signal does not cause a state transition to occur then it is simply discarded (referred to as an *implicit transition*) and the next input signal is considered. During a transition the local variables can be assigned new values and the current value held by variables can be tested.

The behaviour of a process can be specified graphically using a flow graph notation which describes the transition between states, the consumption of input signals and the manipulation of local variables. The basic graphical symbols used to specify a processes behaviour are depicted in figure 2. The symbols have the following meaning:

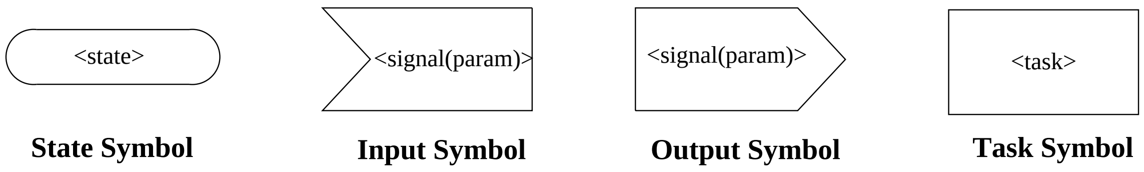

Figure 2: The main graphical symbols for specifying SDL processes.

State Symbol specifies a state <state> for the process;

Input Symbol specifies that a signal <signal> is consumed from the processes input queue and that the variables in the parameter list are set to the corresponding values of the signal;

Output Symbol specifies that a signal <signal> and some associated data values are output;

Task Symbol specifies an assignment for one of the processes local variables within a state transition.

In SDL time is represented by two sorts, *Time* which represents absolute time and *Duration* representing relative time. Both of these sorts are considered to be copies of the real numbers. In any given SDL system it is assumed that there is an absolute global time which all processes in the system can access via the *Now* expression (which always evaluates to the current absolute time). However, since SDL is used to model distributed real–time systems it is assumed that no synchonisation of events in different processes can be based on *Now*. In SDL processes mainly gain access to time via the use of *timers*. A timer can be set by a process to expire at some absolute time (usually defined using the *Now* construct). When a timer expires it places a predefined timeout signal on the input queue of the process which created it. Once a timer has been set it is said to be *active* and it remains active until either it has expired and its timeout signal has been consumed, or it is reset. When a timer is reset any timeout signal it has generated is removed from the processes input queue. We note that setting a timer necessarily involves first resetting the timer. Since a reset removes any existing timeout signals from the processes input queue we know that only one timeout signal from a particular timer can ever be in a processes input queue at any one time.

SDL has many more advanced features for describing the behaviour of processes, including constructs for branching, saving signals, procedures, and dynamically creating/terminating processes. For brevity we omit these from the present discussion.

As an example of using SDL consider the following simple SDL specification of the so called bump game (see CCITT [1988] and Belina et al [1991]). The bump game is a simple computing system which consists of an atomic block B containing two processes $P1$ and $P2$, and five signal routes $sr_1, \ldots, sr_5$. The block B interacts with the environment via the channels *IN1*, *IN2* and *OUT1*. A graphical representation of the SDL system specification of the game is depicted in figure 3, while the associated process specifications are presented in figure 4. The idea is that process $P1$ receives *Bump* signals from the

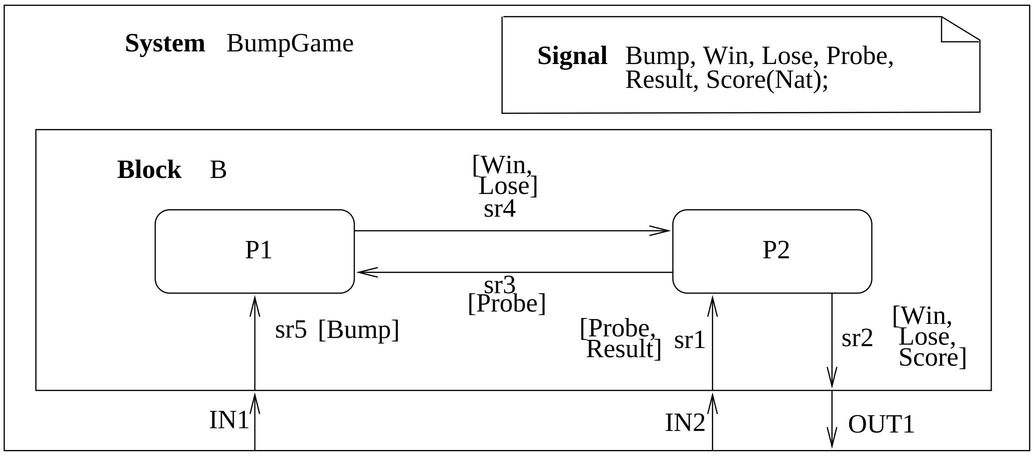

Figure 3: An SDL system diagram for the bump game.

environment via signal route $sr5$ and records if the number of bump signals seen so far is odd or even. Process $P2$ interacts with the player; signal route $sr1$ carries input signals from the player and $sr2$ carries output signals to the player. The player is allowed to guess when he/she thinks the number of bump signals received is odd (this is done by sending a *Probe* signal). Process $P2$ then communicates with process $P1$; it sends a *Probe* signal via $sr3$ and process $P1$ then returns a signal, either *Win* or *Lose*, via $sr4$ indicating if the players guess was correct or not. Process $P2$ then returns *Win* if the player was right (i.e. there has been an odd number of bump signals) otherwise it returns *Lose*. Process $P2$ keeps the score in a local variable *Cnt*, adding on one if the player guesses correctly and taking one off if the player is wrong. The player can ask to be informed of his/her score by sending a *Result* signal to $P2$, which then returns a *Score* signal with the current score as a parameter. A timer *Reset* is used to set the allowed length of time T that can pass between *Probe* signals before the game is reset. If a *Reset* timer signal is read in any state (denoted by the $*$) process P2 restarts the game resetting the players score to zero. Each time a probe signal is read the timer is reset.

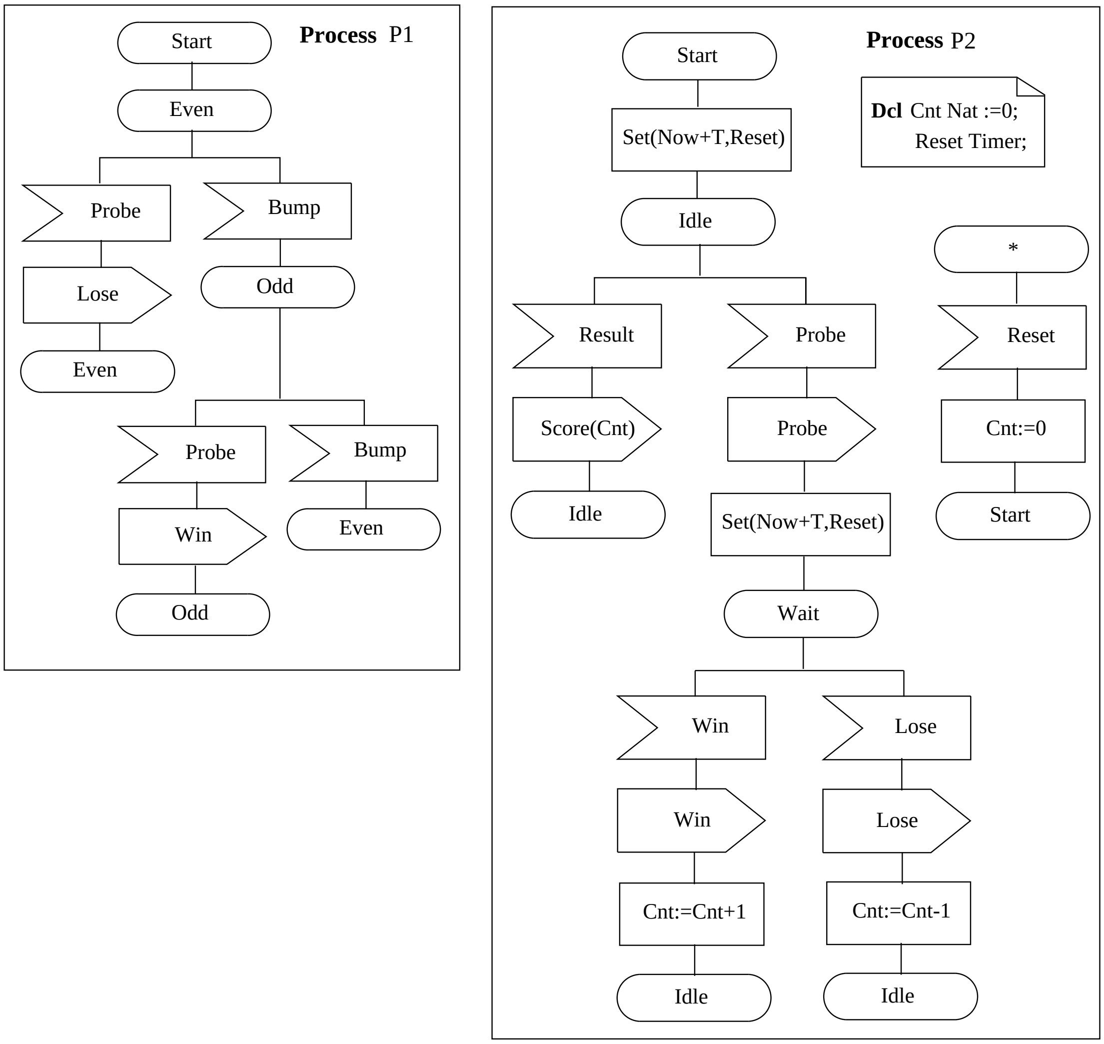

Figure 4: Definition of process $P1$ and $P2$ for the bump game.

4 Modelling SDL Specifications using TRL.

In this section we propose a semantics for SDL specifications based on TRL. We formulate a general approach to modelling SDL specifications and in particular, consider modelling time and timers. The approach we develop will be demonstrated in the next section when we construct a Timed Maude specification for the bump game introduced in Section 3.

An SDL specification can be viewed as consisting of two parts: a static part defining the systems physical structure and data types; and a dynamic part which defines the systems behaviour. The static part of an SDL specification will be modelled using the standard algebraic specification methods provided by Maude. Note that abstract data types in SDL are already defined using algebraic techniques and thus can be straightforwardly coded into Maude. The dynamic part of an SDL specification will be modelled using rewrite rules.

At the lowest level an SDL specification defines process types which taken collectively specify the overall behaviour of the system. Consider an SDL process type P which can be in states $S1, \ldots, Sn$ and has local variables $x_1, \ldots, x_m$ of type $\tau_1, \ldots, \tau_m$. Then to model P we introduce a new class ProP and a new sort StateP with constants $S1, \ldots, Sn$. Process P will be modelled in Maude using two objects $P \otimes Q_P$, where P is the processes main body and Q_P represents the processes input queue. (This division is necessary in order to increase opportunities for concurrent rewriting especially when synchronous communication occurs as described below.) The objects have the form

$$P =< p : \mathit{ProP} \mid \mathit{St} : \mathit{Si}, x_1 : u_1, \ldots, x_m : u_m >, \quad Q_P =< p : \mathit{InQ} \mid Q : q >,$$

where p is a unique object identifier representing the process's pid, ProP is the class for processes of type P, and InQ is the class for input queues. The attribute St stores the current state Si of process P, each attribute x_i stores the current value u_i of the corresponding variable, and Q stores the current input queue q for the process.

In SDL processes communicate with other processes and channels by sending/receiving signals using signal routes. A signal sent using explicit addressing consists of three parts: a destination process pid (process identification number); a signal name; and the pid of the sending process. In order to model signals we use two new sorts Signal and $\mathit{Signame}$, and define a function

$$\mathit{sge} : \mathit{OId} \times \mathit{Signame} \times \mathit{OId} \rightarrow \mathit{Signal},$$

where $\mathit{sge}(p, s, d)$ represents signal s being sent to process d by process p. Signals sent using implicit addressing are modelled using a function

$$\mathit{sg} : \mathit{OId} \times \mathit{Signame} \rightarrow \mathit{Signal},$$

where $\mathit{sg}(p, s)$ represents a signal s sent by process p. For brevity we often denote a signal $\mathit{sg}(p, s)$ by simply s when the identity of the sender is known to be unimportant. We do not explicitly represent signal routes within our model. Instead they are implicitly modelled by restricting the allowed synchronous communication between objects.

The dynamic behaviour of a process is modelled using rewrite rules. Each possible state transition for a process type is modelled by a rewrite rule of the form

$$\begin{array}{l} < p1 : \mathit{ProP1} \mid \mathit{St} : S, x_1 : u_1, \ldots, x_m : u_m > \otimes < p1 : \mathit{InQ} \mid Q : s.q1 > \\ \quad \otimes < p2 : \mathit{InQ} \mid Q : q2 > \longrightarrow \\ < p1 : \mathit{ProP1} \mid \mathit{St} : S', x_1 : u_1', \ldots, x_m : u_m' > \otimes < p1 : \mathit{InQ} \mid Q : q1 > \\ \quad \otimes < p2 : \mathit{InQ} \mid Q : q2.s' >, \end{array}$$

which states that if a process of type $P1$ is in state S and reads signal s then it can perform a transition to state S', possibly altering the values of its local variables (as specified within a task symbol), and outputs a signal s' to process $p2$. In order for the above rule to be valid $p1$ and $p2$ must be in the same atomic block and must be connected by an appropriate signal route. Note that in our Maude model we use only synchronous rules (see Section 2.1.2) for communication between objects (i.e. sending object must synchronise with the input queue of the receiving object) and that Maudes message passing system is not used. One of the reasons for this is to ensure that signals sent along signal routes are transmitted without delay. We note that the use of synchronous rules does not restrict concurrency of the system; if a process object reads a queue of another process it does not block the other process object only its input queue object.

In SDL any signal s on the top of a processes input queue which does not initiate a state transition in a state S is simply discarded. We add the following *standard discard rule* for each discarded signal s in a state S

$$\begin{array}{l} < p1 : \mathit{ProP1} \mid \mathit{St} : S > \otimes < p1 : \mathit{InQ} \mid Q : s.q > \longrightarrow \\ < p1 : \mathit{ProP1} \mid \mathit{St} : S > \otimes < p1 : \mathit{InQ} \mid Q : q > . \end{array}$$

Since it is straightforward to derive these rules for a process we generally omit them in the sequel.

An atomic block B containing processes $P1, \ldots, Pn$ is modelled by an object

$$B = < b : \mathit{Blk} \mid Ps : P1 \otimes Q1 \otimes \cdots \otimes Pn \otimes Qn >,$$

where Blk is the class for blocks, b is a unique block identifier and the attribute Ps stores the current configuration of processes in the block. Communication between blocks occurs along channels which we model explicitly by an object

$$C = < c : \mathit{Chan} \mid Q : q >,$$

where Chan is the class for channels, c is a unique object identifier and Q is an attribute storing the current signals which are being transmitted. Signals are passed between channels and their associated blocks by formulating appropriate rewrite rules for synchronous communication. Note that in our model we restrict channels to being unidirectional to simplify their representation. The non–deterministic delay associated with sending signals along channels is automatically incorporated by the non–deterministic nature of applying rewrite rules. A general block B consisting of sub–blocks $B1, \ldots, Bn$ and channels $C1, \ldots, Cm$ is simply modelled by an appropriate object of type Blk, i.e.

$$B = < b : \mathit{Blk} \mid Ps : C1 \otimes \cdots \otimes Cm \otimes B1 \otimes \cdots \otimes Bn > .$$

Maude provides a module system that allows a specification to be constructed hierarchically. We use this module system to structure our specification of an SDL system by constructing an object module for each block in the system. Each block module then imports the subblock modules it is based on. This approach allows for the reuse of block specifications since Maude provides a range of renaming and parameterization operations for facilitating the reuse of module specifications.

Finally, a system Sys containing at the top level blocks $B1, \ldots, Bn$, input channels $I1, \ldots, Im$ and output channels $O1, \ldots, Ok$ is modelled by a configuration of the form

$$I1 \otimes \cdots \otimes Im \otimes B1 \otimes \cdots \otimes Bn \otimes O1 \otimes \cdots \otimes Ok.$$

The system configuration evolves by concurrently applying the rewrite rules derived for the blocks, channels and processes contained within the original SDL specification.

It is straightforward to incorporate many of the more advanced features of SDL into this model such as saving signals and decision constructs. As an illustrative example we consider how process creation and termination can be modelled.

In SDL a process can create new processes of any type contained within the same block but a process can only be terminated on its own command. The graphical SDL specification for process creation and termination is depicted in figure 5. In this figure part (a) specifies that a process $P1$ creates a new process of type $P2$ after receiving a signal $s1$ in state *S1*. Part (b) of the figure specifies that process $p1$ terminates itself after consuming a signal $s2$ in state *S2*.

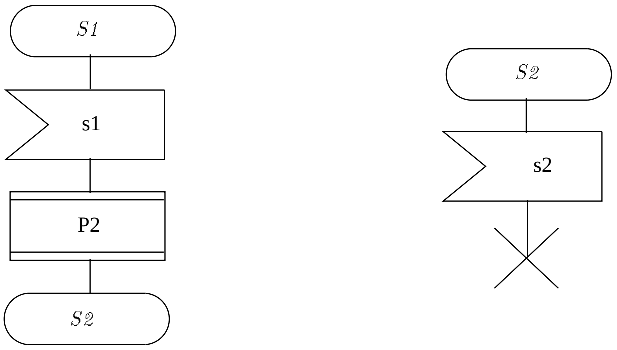

Figure 5: (a) Process create construct. (b) Process termination construct.

The creating process $p1$ and the newly created process $p2$ are represented by the objects

$$< p1 : P1 \mid St : s, \mathit{OffS} : v > \otimes < p1 : \mathit{InQ} \mid Q : q >,$$

$$< i : P2 \mid St : start,\ Pt : u > \otimes < i : \mathit{InQ} \mid Q : empty >$$

where *OffS* stores the pid for the last process created by $p1$ and Pt stores the pid of the processes creator. The block $B1$ which contains $p1$ and $p2$ is represented by $< \mathit{b1} : \mathit{Blk} \mid PC : i, Ps : cf >$, where the attribute PC stores the pid i for the next process to be created in the block. Note that each block B has a constant 0_B associated with it and that i will have the form $i = nxt^k(0_B)$, for some natural number k. This approach allows each block to generate unique pid's for new processes. We can model processs creation by the following rules

$$\begin{array}{l} < \mathit{b1} : \mathit{Blk} \mid PC : i, Ps :< p1 : P1 \mid St : \mathit{S1},\ \mathit{OffS} : v > \otimes \\ \quad < p1 : \mathit{InQ} \mid Q : s1.q1 > \otimes cf > \longrightarrow \\ < \mathit{b1} : \mathit{Blk} \mid PC : nxt(i), Ps :< p1 : P1 \mid St : \mathit{S2},\ \mathit{OffS} : i > \otimes < p1 : \mathit{InQ} \mid Q : q1 > \\ \quad \otimes < i : P2 \mid St : start, Pt : p1 > \otimes < i : \mathit{InQ} \mid Q : empty > \otimes cf >, \end{array}$$

where cf is a variable of type $Conf$. We model process termination using the rule

$$< p1 : P1 \mid St : \mathit{S1}, \mathit{OffS} : v > \otimes < p1 : \mathit{InQ} \mid Q : s1.q > \longrightarrow null,$$

where *null* represents the empty configuration.

In order to represent time within our formal model we use an extension of RL called *timed rewriting logic* (TRL) (see Kosiuczenko and Wirsing [1997]). In TRL the time taken to perform a rewrite rule can be specified and this allows us to reason about the passage of time within a system. To simplify our discussion of time we choose to use discrete time in the sequel and thus we think of the sorts *Time* and *Duration* as being the natural numbers. In fact, as discussed in Bergstra and Middelburg [1995], there are a number of strong reasons motivating the choice of discrete time. (For a treatment of dense time within the Maude framework we refer the interested reader to Kosiuczenko and Wirsing [1997].)

The concept of an absolute global time can be modelled in TRL using a clock object which contains an attribute *time* of type *Time* representing the current absolute time. We then have the rule

$$< c : Clk \mid time : t > - \; r \rightarrow < c : Clk \mid time : t + r >,$$

for each ground term r of type *Time* modelling the passage of time. This clock object could then be added to the system configuration allowing processes within the system to access the global time. However, in SDL it is a fundamental assumption that the global time cannot be used to synchronise processes. Infact, in practice the global time is normally only used to set timers to expire in some relative time via the *Now* construct. For this reason we have chosen not to explicitly model a global time but instead to allow timers to act as counters which produce a timeout signal after a specified period of time (see Kosiuczenko and Wirsing [1997]). We model timers using objects of the form

$$< tm : Tm \mid time : t, P : p, TO : b >,$$

where tm is of sort *Timer* (which is a subsort of both *OId* and *Signal*) and Tm is the class of timer objects. In the above object timer tm has t time units left before it outputs a timeout signal tm to it's creating process p. The attribute TO indicates if a timeout signal has been sent or not (this is needed since a timer remains active after sending a timeout signal). The names of the timers which are currently active for a process P are stored in a queue within the process using an attribute TS (we will see in the next section that this extra information is needed to axiomatise the resetting of timers).

The following are general rules which model the passage of time for timers, what happens when a timer expires, and the passage of time once a timeout signal has been sent by a timer.

$$< tm : Tm \mid time : t + r > - \; r \rightarrow < tm : Tm \mid time : t >,$$

$$\begin{array}{l} < tm : Tm \mid time : 0, P : p, TO : False > \otimes < p : InQ \mid Q : q > - \; 0 \rightarrow \\ < tm : Tm \mid time : 0, P : p, TO : True > \otimes < p : InQ \mid Q : q.tm >, \end{array}$$

$$< tm : Tm \mid time : 0, TO : True > - \; r \rightarrow < tm : Tm \mid time : 0, TO : True > .$$

Note that once we introduce time in to our model we have to substitute timed rewrite rules for all the ordinary rewrite rules we have so far used (although we note that a range of possible times can be associated with a transition, see Kosiuczenko an Wirsing [1995]). This allows us to specify a bound on the time taken to perform each action or transition in the system. At present SDL does not provide a means of specifying this information. However, such timing constraints become important when reasoning about the timing properties of an SDL system and for this reason we feel an appropriate extension to SDL is needed. The above approach also allows processes to have different clock rates, as is indeed often the case in real distributed systems. All that would be needed is to change the increment rate in the first rule presented above (see Kosiuczenko and Wirsing [1995]).

5 Example: The Bump Game.

In this section we consider applying the approach developed in the previous section to produce a TRL model for the SDL specification of the bump game presented in Section 3. We begin by considering the two processes $P1$ and $P2$ which we model using the following objects

$$\mathit{P1} =< p1 : \mathit{Pro1} \mid \mathit{St} : S1 >, \quad \mathit{Q1} =< p1 : \mathit{InQ} \mid Q : q1 > \text{ and}$$

$$\mathit{P2} =< p2 : \mathit{Pro2} \mid \mathit{St} : S2, \ \mathit{TS} : tq, \ \mathit{Cnt} : c >, \quad \mathit{Q2} =< p2 : \mathit{InQ} \mid Q : q2 >,$$

where the object names $p1$ and $p2$ are unique process identification numbers, *Pro1* and *Pro2* are the classes representing processes of type $P1$ and $P2$ respectively, and *InQ* is the class for input queues. The attribute *TS* holds a list of active timers and *Cnt* represents the local variable which stores the players current score. The two processes communicate with each other and the external channels using the five signal routes $sr_1, \ldots, sr_5$. These signal routes are modelled by restricting the allowed synchronous communication between process and channel objects.

Next we define an object to represent the atomic block B which contains the two processes

$$B =< b : \mathit{Blk} \mid \mathit{Ps} : \mathit{P1} \otimes \mathit{Q1} \otimes \mathit{P2} \otimes \mathit{Q2} > .$$

The object contains a multiset consisting of the processes *P1* and *P2* as part of its internal state. The block has three channels associated with it, two input channels *IN1* and *IN2*, and one output channel *OUT1*. We model these by three distinct objects

$$\mathit{IN1} =< \mathit{in1} : \mathit{Chan} \mid Q : q1 >, \ \mathit{IN2} =< \mathit{in2} : \mathit{Chan} \mid Q : q2 > \text{ and}$$

$$\mathit{OUT1} =< \mathit{out1} : \mathit{Chan} \mid Q : q3 >,$$

where *in1*, *in2* and *out1* are unique object identifiers.

The complete system will be represented by the multiset

$$\mathit{IN1} \otimes \mathit{IN2} \otimes B \otimes \mathit{OUT1}.$$

Using our proposed approach we can formulate the following Timed Maude specification for the bump game. Note that object modules in Maude provide the predefined sorts *Conf* of configurations (multisets) and *OId* of object identifiers (see Meseguer [1993]).

We begin with a module *SDLBase* for the basic types and definitions needed when modelling an SDL specification. For brevity we have omitted the rules introduced in Section 4 for timers.

omod SDLBase **is**
  Sorts
    PId, *Que*, *Signame*, *Timer*, *Signal*.

Subsorts

$PId < OId$. $Signame < Signal < Que$.
$Timer < Signal$. $Timer < OId$.

Constants

$Env : PId$.
$empty : Que$.

Functions

$nxt : PId \rightarrow PId$.
$sg : Signame \times OId \rightarrow Signal$.
$\_.\_ : Que \times Que \rightarrow Que \quad [assoc \;\; id : empty]$.

Classes

$InQ \mid Q : Que$.
$Blk \mid Ps : Conf$.
$Chan \mid Q : Que$.
$Tm \mid time : Time, P : PId, TO : Bool$.

endom

In order to axiomatise the operation of the timer *Reset* we need to introduce two auxilary functions: $Active : Timer \times Que \rightarrow Bool$ which tells us if a timer is active or not; and $Rem : Timer \times Que \rightarrow Que$ which removes a timer signal from a queue of signals or timers. These functions are axiomatised by the following equations:

$$Active(tm, empty) = False,$$

$$Active(tm, tm1.tq) = if \;\; eq(tm, tm1) \; then \; True \; else \; Active(tm, tq),$$

$$Rem(tm, empty) = empty,$$

$$Rem(tm, s.q) = if \;\; eq(tm, s) \; then \; Rem(tm, q) \; else \; s.Rem(tm, q),$$

where $eq : Signal \times Signal \rightarrow Bool$ is the characteristic function for the equality relation on signals (axiomatised in the obvious way). These definitions should be added to the Maude module *SDLBase* above.

We can now use the above base module to define a module *BumpGame* for the complete system. (Note we assume the existence of a functional module *Nat* specifying the natural numbers.) We encourage the reader to compare the structure of this specification with the SDL specification depicted in figures 3 and 4.

omod BumpGame **is**

Protecting Nat.
Extending $SDLBase$.
Sorts

$StateP1, StateP2$.

Constants

$p1, p2 : \mathit{PId}.$
$\mathit{Result}, \mathit{Win}, \mathit{Lose}, \mathit{Probe}, \mathit{Bump} : \mathit{Signame}.$
$\mathit{Reset} : \mathit{Timer}.$
$\mathit{start}, \mathit{odd}, \mathit{even} : \mathit{StateP1}.$
$\mathit{start}, \mathit{idle}, \mathit{wait} : \mathit{StateP2}.$
$b, \mathit{in1}, \mathit{in2}, \mathit{out1} : \mathit{OId}.$
$T : \mathit{Time}.$

Functions

$\mathit{Score} : \mathit{Nat} \rightarrow \mathit{Signame}.$

Classes

$\mathit{Pro1} \mid \mathit{St} : \mathit{StateP1}.$
$\mathit{Pro2} \mid \mathit{St} : \mathit{StateP2},\ \ \mathit{TS},\ \ \mathit{Cnt} : \mathit{Nat}.$

Variables

$c : \mathit{Nat}.$
$\mathit{bl} : \mathit{Bool}.$
$t : \mathit{Time}.$
$s : \mathit{Signal}.$
$\mathit{cf} : \mathit{Conf}.$
$\mathit{st} : \mathit{StateP2}.$
$q, q1, q2, \mathit{tq} : \mathit{Que}.$

Rules

(* Rules for Process 1 *)
(* Initial transition from Start state *)
$< p1 : \mathit{Pro1} \mid \mathit{St} : \mathit{start} > -\ 1 \rightarrow < p1 : \mathit{Pro1} \mid \mathit{St} : \mathit{even} > .$

(* Bump signal consumed in Even state *)
$< p1 : \mathit{Pro1} \mid \mathit{St} : \mathit{even} > \otimes < p1 : \mathit{InQ} \mid Q : \mathit{Bump}.q > -\ 1 \rightarrow$
$< p1 : \mathit{Pro1} \mid \mathit{St} : \mathit{odd} > \otimes < p1 : \mathit{InQ} \mid Q : q > .$

(* Bump signal consumed in Odd state *)
$< p1 : \mathit{Pro1} \mid \mathit{St} : \mathit{odd} > \otimes < p1 : \mathit{InQ} \mid Q : \mathit{Bump}.q > -\ 1 \rightarrow$
$< p1 : \mathit{Pro1} \mid \mathit{St} : \mathit{even} > \otimes < p1 : \mathit{InQ} \mid Q : q > .$

(* Probe signal loses as even number of Bump signals *)
$< p1 : \mathit{Pro1} \mid \mathit{St} : \mathit{even} > \otimes < p1 : \mathit{InQ} \mid Q : \mathit{Probe}.q >$
$\quad \otimes < p2 : \mathit{InQ} \mid Q : q1 > -\ 1 \rightarrow$
$< p1 : \mathit{Pro1} \mid \mathit{St} : \mathit{even} > \otimes < p1 : \mathit{InQ} \mid Q : q > \otimes < p2 : \mathit{InQ} \mid Q : q1.\mathit{Lose} > .$

(* Probe signal wins as odd number of Bump signals *)
$< p1 : \mathit{Pro1} \mid \mathit{St} : \mathit{odd} > \otimes < p1 : \mathit{InQ} \mid Q : \mathit{Probe}.q >$
$\quad \otimes < p2 : \mathit{InQ} \mid Q : q1 > -\ 1 \rightarrow$

$< p1 : Pro1 \mid St : odd > \otimes < p1 : InQ \mid Q : q > \otimes < p2 : InQ \mid Q : q1.Win > .$

(* Rules for Process 2 *)
(* Initial transition from Start state which sets timer *)

$$< p2 : Pro2 \mid St : start, TS : tq > \otimes < p2 : InQ \mid Q : q > \otimes$$
$$< Reset : Tm \mid time : t, P : p2, TO : bl > -1 \rightarrow$$
$$< p2 : Pro2 \mid St : idle, TS : tq > \otimes < p2 : InQ \mid Q : Rem(Reset, q) > \otimes$$
$$< Reset : Tm \mid time : T, P : p2, TO : False >$$
$$if\ Active(Reset, tq) = True.$$

$$< p2 : Pro2 \mid St : start, TS : tq > -1 \rightarrow$$
$$< p2 : Pro2 \mid St : idle, TS : tq.Reset > \otimes$$
$$< Reset : Tm \mid time : T, P : p2, TO : False > \quad if\ Active(Reset, tq) = False.$$

(* Probe signal consumed so send Probe signal to P1 and reset timer *)

$$< p2 : Pro2 \mid St : idle, TS : tq > \otimes < p2 : InQ \mid Q : Probe.q > \otimes$$
$$< Reset : Tm \mid time : t, P : p2, TO : bl > \otimes < p1 : InQ \mid Q : q1 > -1 \rightarrow$$
$$< p2 : Pro2 \mid St : wait, TS : tq > \otimes < p2 : InQ \mid Q : Rem(Reset, q) > \otimes$$
$$< Reset : Tm \mid time : T, P : p2, TO : False > \otimes < p1 : InQ \mid Q : q1.Probe >$$
$$if\ Active(Reset, tq) = True.$$

$$< p2 : Pro2 \mid St : idle, TS : tq > \otimes < p2 : InQ \mid Q : Probe.q >$$
$$\otimes < p1 : InQ \mid Q : q1 > -1 \rightarrow$$
$$< p2 : Pro2 \mid St : wait, TS : tq.Reset > \otimes < p2 : InQ \mid Q : q > \otimes$$
$$< Reset : Tm \mid time : T, P : p2, TO : False > \otimes < p1 : InQ \mid Q : q1.Probe >$$
$$if\ Active(Reset, tq) = False.$$

(* Reset timer signal consumed in any state, so set Cnt to zero *)

$$< p2 : Pro2 \mid St : st,\ Cnt : c > \otimes < p2 : InQ \mid Q : Reset.q > -1 \rightarrow$$
$$< p2 : Pro2 \mid St : start,\ Cnt : 0 > \otimes < p2 : InQ \mid Q : q > .$$

(* Result signal consumed so send score Cnt to output channel *)

$$< out1 : Chan \mid Q : q1 > \otimes < b : Blk \mid Ps :< p2 : Pro2 \mid St : idle,\ Cnt : c >$$
$$\otimes < p2 : InQ \mid Q : Result.q > \otimes cf > -1 \rightarrow$$
$$< out1 : Chan \mid Q : q1.Score(c) > \otimes$$
$$< b : Blk \mid Ps :< p2 : Pro2 \mid St : idle,\ Cnt : c > \otimes < p2 : InQ \mid Q : q > \otimes cf > .$$

(* Win signal consumed so send Win signal and add one to Cnt *)

$$< out1 : Chan \mid Q : q1 > \otimes < b : Blk \mid Ps :< p2 : Pro2 \mid St : wait,\ Cnt : c > \otimes$$
$$< p2 : InQ \mid Q : Win.q > \otimes cf > -1 \rightarrow$$
$$< out1 : Chan \mid Q : q1.Win > \otimes\ < b : Blk \mid Ps :< p2 : Pro2 \mid St : idle,$$
$$Cnt : succ(c) > \otimes < p2 : InQ \mid Q : q > \otimes cf > .$$

(* Lose signal consumed so send Lose signal and decrease Cnt by one *)

$$< out1 : Chan \mid Q : q1 > \otimes < b : Blk \mid Ps :< p2 : Pro2 \mid St : wait,\ Cnt : c >$$
$$< p2 : InQ \mid Q : Lose.q > \otimes cf > -1 \rightarrow$$

$< \mathit{out1} : \mathit{Chan} \mid Q : q1.\mathit{Lose} > \otimes \; < b : \mathit{Blk} \mid \mathit{Ps} :< p2 : \mathit{Pro2} \mid \mathit{St} : \mathit{idle},$
$\quad \mathit{Cnt} : \mathit{pred}(c) > \otimes < p2 : \mathit{InQ} \mid Q : q > \otimes cf > .$

(* Rules for input from environment *)
(* Send signals on channel IN1 to process P1 *)
$< \mathit{in1} : \mathit{Chan} \mid Q : s.q1 > \otimes < b : \mathit{Blk} \mid \mathit{Ps} :< p1 : \mathit{InQ} \mid Q : q > \otimes cf > -\, 1 \rightarrow$
$< \mathit{in1} : \mathit{Chan} \mid Q : q1 > \otimes < b : \mathit{Blk} \mid \mathit{Ps} :< p1 : \mathit{InQ} \mid Q : q.s > \otimes cf > .$

(* Send signals on channel IN2 to process P2 *)
$< \mathit{in2} : \mathit{Chan} \mid Q : s.q1 > \otimes < b : \mathit{Blk} \mid \mathit{Ps} :< p2 : \mathit{InQ} \mid Q : q > \otimes cf > -\, 1 \rightarrow$
$< \mathit{in2} : \mathit{Chan} \mid Q : q1 > \otimes < b : \mathit{Blk} \mid \mathit{Ps} :< p2 : \mathit{InQ} \mid Q : q.s > \otimes cf > .$

endom

In order to complete the above specification we neeed to add standard discard rules (see Section 4) for process $P2$ when it is in state *idle* and it reads signal *Win* or *Lose*, and state *wait* when it reads signal *Result* or *Probe*.

The above Timed Maude specification gives a precise and natural formal semantics to the SDL specification of the Bump game which can be used for simulating, testing and verifying properties of the system. It also provides a formal basis for analysing the timing properties of the Bump game system. In the above specification we have chosen a simple fixed time consumption pattern for the system, i.e. zero time for implicit transitions and one time unit for all other state transitions. We note that other time consumption patterns representing different implementation constraints could be used and the resulting new timing properties investigated using TRL.

6 Concluding Remarks.

In this paper we have presented a comprehensive formal semantics for SDL based on the new algebraic formalism of Timed Rewriting Logic. This new semantics has a number of key advantages over its predecessors, including: a natural correspondence between the structure of the semantics and the corresponding SDL constructs; integration of the static and dynamic views of an SDL specification within a single unifying formalism; and the use of a new object–oriented specification language, Timed Maude, with its associated support tools.

In future work we intend to consider extending our semantics to the object–oriented features of SDL–92 (see see Faergemand and Olsen [1994]) making further use of Maude's concepts of object, classes, and inheritance. Though our semantics allows us to describe real-time systems, its operational style makes it difficult to express more complex real-time requirements. It therefore may be reasonable to consider in future work combining our approach with temporal logic (see Leue [1995] and Mork et al [1996]) and to consider extending the SDL syntax accordingly. Finally we note that SDL specifications are closely related to message sequence charts (MSCs) (see for example Rudolph et al [1996]). In future work we intend to investigate the relationship between these two real–time FDTs using the formal semantics presented in this paper and the work developed in Kosiuczenko [1997].

Acknowledgements.
It is a pleasure to thank K. Meinke and M. Wirsing for their helpful comments and advice during the preparation of this paper. We are also very grateful to U. Hinkel for many interesting and informative discussions concerning SDL. Finally, we would like to grateful acknowledge the financial support of the British Council and DAAD which has made this collaborative work possible.

7 References.

F. Belina and D. Hogrefe. The CCITT Specification and Description Language SDL. *Computer Networks and ISDN Systems*, 16:311–341, 1989.

F. Belina, D. Hogrefe and A. Sarma. *SDL with Applications from Protocol Specification.* Prentice Hall International, 1991.

J. A. Bergstra and C. Middelburg. Process Algebra Semantics φSDL. In: Proc. of ACP '95, The Second Workshop on Algebra of Communicating Process, Eindhoven University of Technology, Department of Mathematics and Computing Science, pages 309–346, Report No. 95-14, 1995.

M. Broy. Towards a formal foundation of the specification and description language SDL. *Formal Aspects of Computing*, 3:21–57, 1991.

CCITT. *Recommendation Z. 100 – Functional Specification and Description Language (SDL), BLUE BOOK, Fascile X.1 and X.5*, Volume X. International Telecommuncation Union, 1988.

CCITT. *Revised Recommendation Z. 100 – CCITT Specification and Description Language (SDL), COM X–R 26*, Geneva, May 1992.

H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1 – Equations and Initial Semantics.* EATCS Monographs on Theoretical Computer Science 6. Springer-Verlag, Berlin, 1985.

O. Færgemand and A. Olsen. Introduction to SDL–92. *Computer Networks and ISDN Systems*, 26:1143–1167, 1994.

U. Hinkel. *A formal semantics for SDL based on FOCUS.* Ph. D. Thesis, Technical University Munich, 1997. (In preparation.)

E. Holz and K. Stolen. An Attempt to Embed a Restricted Version of SDL as a Target Language in Focus. In: D. Hogrefe and S. Leue (eds), *Formal Description Techniques VII*, Chapman and Hall, 1995.

P. Kosiuczenko and M. Wirsing. Timed Rewriting Logic with Applications to Time-Sensitive Systems. To appear in: H. Schwichtenberg (ed.), *Proceedings of the International Summer School on Proof and Computation*, Marktoberdorf 1995.

P. Kosiuczenko and M. Wirsing. Timed rewriting logic with an application to object-based specification. *Science of Computer Programming*, 28:225–246, 1997.

P. Kosiuczenko. A Complete Semantics of Basic Message Sequence Charts: An Algebriac Approach. Technical Report, Institut für Informatik, Ludwig–Maximilians–Universität, Munich, 1997. (To appear.)

S. Leue. Specifying Real-Time Requirements for SDL Specifications - A Temporal Logic-Based Approach. In: *Procs. of the Fifteenth International Symposium on Protocol Spec-*

ification, Testing, and Verification PSTV'95, Chapmann and Hall, 1995.
J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In: G. Agha, P. Wegner and A. Yonezawa (eds), *Research Directions in Concurrent Object–Oriented Programming*, MIT Press, 1993.
J. Meseguer and T. Winkler. Parallel programming in Maude. In: J.-P. Banâtre and D. Le Mètayer (eds), *Research Directions in High–Level Parallel Programming Languages*, Lecture Notes in Computer Science 574, pages 253–293, Springer–Verlag, 1992.
S. Mork, J. Godskesen, M. Hansen and R. Sharp. A Timed Semantics for SDL. In R. Gotzhein and J. Bredereke (eds), *Formal Description Techniques IX*, Chapman and Hall, 1996.
P. Olveczky, P. Kosiuczenko, and M. Wirsing. Steamboiler specification problem: an algebraic object-oriented solution. In: J. R. Abrial, E. Boerger, H. Lnagmaack (Eds.), *Formal Methods for Industrial Applications*, Lecture Notes in Computer Science 1165, Springer–Verlag, 1996.
E. Rudolph, P. Graubmann and J. Grabowski. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28:1629–1641, 1996.
M. Wirsing. Algebraic specification. In: J. van Leeuwen (ed) *Handbook of Theoretical Computer Science*, Vol. B, pages 675–788, North Holland, Amsterdam, 1990.