variable values for signals and not expressed their interrelationships. A conditional split must then be introduced into the behaviour of a program to allow for both possibilities (i.e. where one branch assumes an event and the other does not). The user may of course choose at this stage to specialise the variables being used to ensure either one of these possible behaviours. If not, `finish_GAMMA` introduces the appropriate split. Examples of the use of `finish_GAMMA` in context are provided in Chapter 6.

The use of `finish_GAMMA` emphasises one particularly troublesome feature of the animation of not only Femto-VHDL programs, but also of full VHDL. If care is not taken in the specification of initial values of signals, it is quite possible to engender this kind of conditional split for every signal almost every time around the simulation loop. The result is naturally an explosion in the size of the terms being examined. It also puts one in the situation of doing exhaustive search over all possible values for the signals, which is exactly the kind of reasoning that is trying to be avoided.

From the point of view of automation, the case splits engendered by `finish_GAMMA` are one of the reasons behind not attempting to construct a completely automatic tool. Such choice points may be of interest in individual circumstances where one particular course of action needs to be taken. If, however, one was not interested in human intervention, it would be simple to write a naïve conversion to repetitively go around the simulation loop simply by composing `ONCE_AROUND` and `finish_GAMMA`. The problem is, of course, that the simulation may not quiesce. In those situations, the new conversion would loop infinitely.

The three parts of the cycle, "move to nearest interesting point", "update current state" and "calculate events", are animated by `simp_state_CONV`. The conversion first traverses the list of delays trying to determine the next point of computation. Having ascertained that time, the posted transactions for that time are extracted. These transactions are then compared to the existing signal values to give a new set of signal-value pairs and a new set of events via `NEXT_SIGMA_CONV` and `NEXT_GAMMA_CONV` respectively. Once the new environment has been derived in this fashion, the final part of the diagram ("perform a simulation cycle") is entered into. Here `RUNCYCLE_CONV` is the top level for another set of related conversions which execute the simulation cycle.

Also recall the steps that make up a simulation cycle:

- determine which processes are active based on current events
- run the active processes in parallel
- merge all processes' scheduled transactions into a collective whole

`RUNCYCLE_CONV` first traverses the sea of processes to wake up and run any of them which are sensitive to the just-derived events. This is accomplished by making use of the rules for the simulation cycle as defined in Chapter 4 and embedded in the HOL system via `new_inductive_definition`. Once a process has been activated, the statements inside it are animated by use of `RUNSEQ_CONV` and `RUNBOOL_CONV`, which are responsible for the execution of the rules for sequential statements and Boolean expressions respectively. The behaviour resulting from the activation of any two processes in parallel is resolved from calls to clean_zip to zip by `CLEAN_ZIP_CONV`.

This account of the sequence of conversions hides the fact that simplification of the results arising from many of the steps is also being carried out. A simple example is that calls to clean_zip are reduced to the appropriate sequence of zip's. These calls to zip are also further simplified by the collapsing out of any empty transactions ($\lambda t.\{\}$). Once normalised, the resulting transactions are more concise and therefore make the work of `simp_state_CONV` much easier.

The Conversion `finish_GAMMA`

It is sometimes the case that when `NEXT_GAMMA_CONV` is attempting to ascertain the next set of events, there is not enough information about the state of the signals to ascertain whether or not a new value for particular a signal represents a change from the old one. This situation frequently occurs when the user has supplied

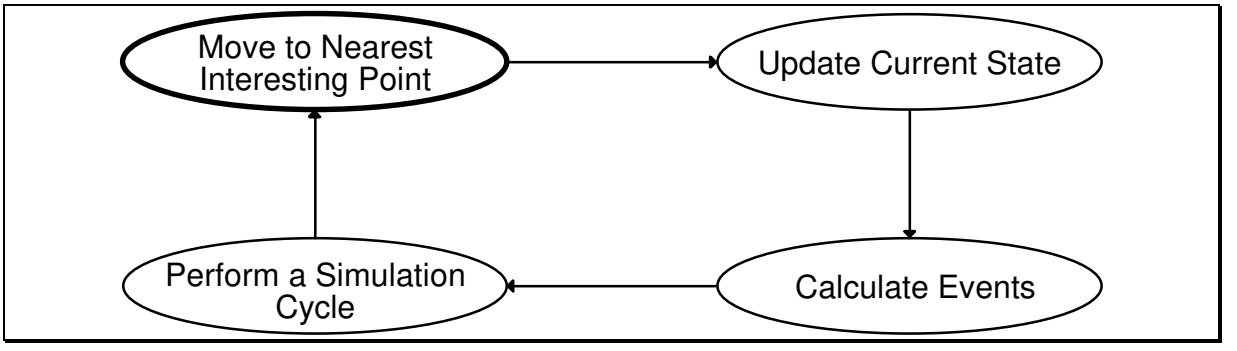
those behaviours or their equivalence to other behaviours may be discerned by proof.

In order to make use of the semantics to derive these behaviours for individual programs, it becomes necessary to write special-purpose conversions to animate the definitions. To a large extent, the process is one of transforming one theorem into another by expanding the definitions that it comprises in an intelligent way. To that end, almost every definition made in the semantics has had a HOL conversion written for it. These conversions are often rewriting engines to replace a definition with its body. Frequently however, it is possible to do some simplification of the result to get a clearer statement (simplification of arithmetic expressions, normalisation, etc.). In these cases the conversions must also do context-dependant reasoning. Most of the time, this leads to faster execution time as it cuts down on the size of the terms being manipulated.

In the course of working with the semantics of individual Femto-VHDL programs, two particular conversions will be frequently used. The first is one that animates one iteration of the simulation loop, and is aptly-enough named `ONCE_AROUND`. The other, `finish_GAMMA`, helps trigger an event when there is insufficient contextual information about signal values. None of the ML code for either of these conversions will be given. Instead the algorithm and rationale behind each will be presented. Examples of their use follow in Chapter 6.

The Conversion `ONCE_AROUND`

Recall the diagram for the top-level simulation loop:



The purpose of `ONCE_AROUND` is to co-ordinate these steps for individual Femto-VHDL programs by using the rules and definitions of the semantics. Each of the actions above represents a suite of additional proof procedures which animate the related definitions.

When given to the defining function, they appear as:

```
let (csrules,csind) =
    let CS="—cyc→:cycenv->CS->trans->trans->bool" in
    new_inductive_definition false `—cyc→_DEF`
    ("^CS ρ Stmnt τ τ'",[])
    [[
      %--------------------------------% "(sl ∩ γ_of_ρ ρ))={}"],
      "^CS ρ (PROCESS sl SS) τ (λt.{})";
     [    "—seq→ ρ SS τ τ'";
      %--------------------------% "~((sl ∩ (γ_of_ρ ρ))={})"],
      "^CS ρ (PROCESS sl SS) τ τ'";
     [    "^CS ρ S0 τ τ'" ; "^CS ρ S1 τ τ''"
      %-----------------------------------% ],
      "^CS ρ (PAR S0 S1) τ (CLEAN_ZIP τ τ' τ'')"]
```

The function γ_of_ρ extracts the γ portion of the environment ρ. The actual constants used for —cyc→ and —seq→ are `SimCycle` and `SeqStmnt` respectively. The arguments to `new_inductive_definition` are much the same as other similar functions in the HOL system. The first argument is a flag to tell the system whether or not the constant being defined is infix or prefix. The second argument is the name associated with the definition in the theory. The third argument is a 'pattern' that supplies information which is needed because the function may be used to define classes of inductively defined relations, rather than just single instances of these relations [25]. The final argument is a list of term list-term pairs, each element of which describes one rule. The term list contains the hypotheses and side conditions. An empty list is used for axioms. The term is the conclusion of the rule. In HOL, any text between pairs of `%` is interpreted as a comment. So, each of the lines separating hypotheses from conclusion are really nothing more than comments