Embedded systems programming: Accessing databases from Esterel

David White and Gerald Lüttgen
Department of Computer Science, University of York
Heslington, York YO10 5DD, UK
{dwhite, luettgen}@cs.york.ac.uk

Abstract

A current limitation in embedded controller design and programming is the lack of database support in development tools such as Esterel Studio. This article proposes a way of integrating databases and Esterel by providing two Application Programming Interfaces (APIs) which enable the use of relational databases inside Esterel programs. As databases and Esterel programs are often executed on different machines, result sets returned as responses to database queries may be processed either locally and according to Esterel's synchrony hypothesis, or remotely along several of Esterel's execution cycles. These different scenarios are reflected in the design and usage rules of the two APIs presented in this article, which rely on Esterel's facilities for extending the language by external data types, external functions and procedures, as well as tasks.

The APIs' utility is demonstrated by means of a case study modelling an automated warehouse storage system, which is constructed using Lego Mindstorms robotics kits. The robot's controller is programmed in Esterel in a way that takes dynamic ordering information and the warehouse's floor layout into account, both of which are stored in a MySQL database.

Keywords: Embedded systems programming, synchronous languages, Esterel, relational databases, API, Lego Mindstorms.

1 Introduction

One of the current limitations in the programming of embedded controllers is the lack of *database support* available within languages such as Esterel [3, 4] and Lustre [9], and their development environments, *Esterel Studio* and *SCADE* [11], respectively. These environments are used by large avionics manufacturers and vendors of digital signal processing solutions for developing the software of complex, and often safety–critical, embedded systems. Both Esterel and Lustre are *synchronous languages* which aim at describing reactions in cycle–based reactive systems, including embedded controllers. Such systems continuously interact with their physical environment by (i) reading in signals representing sensor values, such as an aircraft's speed, altitude and attitude, (ii) computing a reaction based on these values, such as a rudder angle, and (iii) emitting signals carrying the computed values to the environment, e.g., to the hydraulic system moving the rudders. While Esterel is a textual, imperative language that aims at modelling control flow and has semantical similarities to *Statecharts* [21],

Lustre is best suited for modelling data flow and is a graphical language centred around block diagrams, very much like *Simulink* [18].

The problem. What all development environments that are available for these languages have in common is that they support the automatic generation of code, such as C, Ada or VHDL code, from abstract program descriptions. In this way, they aim to make embedded software design and programming more cost–effective when compared to traditional software development processes. However, Esterel Studio and SCADE do not provide an easy way of integrating databases within an application. Other reactive systems design tools are very limited in this respect as well, including *Simulink/Stateflow* [6] and *Statemate* [13, 14]. As is, a system designer needs to modify auto–generated code by hand in order to interface to databases, which is both difficult and error–prone. This is a problem very much relevant in industry since some reactive systems programmed in synchronous languages would benefit from an easy model of database interaction. For example, synchronous languages are often used to build the flight software for aeroplanes. Adding database interaction would enable spacial and mapping data to be retrieved and processed directly by the reactive kernel implementing an auto pilot. Further examples are infotainment systems in the automotive sector, particularly navigation systems, or process control systems in nuclear reactors where regulators require that logs of data are recorded and kept.

Our contribution. This article addresses the aforementioned limitation by providing *Application Programming Interfaces* (APIs) for using relational databases within the Esterel programming language. We choose MySQL [19] as the database and, since reactive kernels are produced as C programs by the Esterel compiler [2, 10, 22], the APIs are implemented using the MySQL C interface [20] whose functionality we aim to mirror in our APIs for Esterel. MySQL is selected here simply for its convenience and since it is widely used. However, our approach can as easily be applied to other relational databases. To the best of our knowledge, no work on database integration within Esterel, or similar languages, has been published in the literature before. This does not mean, however, that we are the first to integrate an existing synchronous language with a database system. The problem is that other works are commercial and not in the open domain. This includes National Instruments' *LabVIEW Database Connectivity Toolkit* [15] which is a set of tools for connecting programs designed in LabVIEW to popular databases, such as Microsoft Access, SQL Server and Oracle, and for implementing many common database operations without having to perform SQL programming.

Because database transactions are relatively complex when compared to responses of Esterel reactive kernels, databases and reactive programs must be considered as running asynchronously to each other. This is true regardless of whether they reside on the same machine or on different machines. In the former case, however, *result sets* to database queries may reasonably be assumed to be processed within a single synchronous step of the reactive kernel. In the latter case, result sets are necessarily read asynchronously to the reactive kernel. For these reasons, one API for each situation is provided: a *Local Result Set API* and a *Remote Result Set API*. The realisation of both APIs relies on Esterel's support for extending the language via external data types, external functions and procedures, and tasks. For example, the Local Result Set API makes heavy use of external functions and procedures. This is because these are considered to execute instantaneously, i.e., within a single reaction cycle, and therefore do not interfere with the synchronous nature of Esterel programs. On

the other hand, the Remote Result Set API is implemented using Esterel's task mechanism which allows external code to run asynchronously to an Esterel program, i.e., alongside several program cycles, but still be controlled by it.

We demonstrate the utility of our APIs by means of a case study involving a *warehouse storage system*. The idea behind this is that of a direct order company, i.e., orders must be picked from items stored in a warehouse. Accordingly, a *robot* is built and programmed to pick up and drop off items, therefore making it possible for a complete order to be collected and stored. The orders and the information about the items are provided by a database. Part of the information stored on an item is its position in the warehouse, thereby providing mapping data for the robot. The case study is realised using *Lego Mindstorms* robotics kits [1, 17], which provide a small programmable brick, called the *RCX*, that houses a microcomputer capable of running an Esterel reactive kernel. Sensors and actuators connected to the RCX, such as *touch*, *light* and *rotation sensors* and *motors*, respectively, permit interaction with the RCX's environment. The RCX also has a built–in infrared port, which we use to communicate with the warehouse database on the server.

Organisation. The next section gives a brief introduction to the Esterel language. Sec. 3 describes both our APIs, emphasising the general model of interaction between Esterel reactive kernels and databases. Some details regarding the APIs' implementations can be found in Sec. 4. Our case study involving the warehouse storage system is presented in Sec. 5, while Sec. 6 contains our conclusions.

2 A brief overview of Esterel

The *Esterel* language for programming embedded controllers has been developed by Gérard Berry in France since the early Eighties [5] and has been commercialised by Esterel Technologies in their *Esterel Studio* design suite [11].

Esterel is part of a family of languages — the so called *synchronous languages* — that are reactive and synchronous. *Reactive* means that an Esterel program is constantly interacting with its environment, while *synchronous* means that these interactions are periodic, with the computation of each reaction being "instantaneous", i.e., reactions are computed faster than the environment is sampled for inputs. Esterel affects and samples its environment through *signals*, which are its primary communication device. For example, a signal representing a button press will either be *present* if the button is pressed or *absent* if not. Signals can also carry data values, which is useful when working with sensor readings and actuator parameters.

Moreover, Esterel is an *imperative* programming language. It is particularly suited for control–dominated applications as it allows for the expression of parallelism and preemption. Its core language elements include the following statements [3]:

- `emit S` emits signal `S` in the current instant, i.e., the current reaction cycle;
- `present S then` *stmt1* `else` *stmt2* `end` checks whether signal `S` can be determined as either present or absent in the current instant; if it is present, then *stmt1* is executed, and if it is absent, then control passes to *stmt2*;
- `pause` stops execution in the current instant, and resumes execution in the next instant;

- `loop` *stmts* `end` executes *stmts* repeatedly; the loop body *stmts* must include at least one `pause` statement;
- *stmt1* `||` *stmt2* runs *stmt1* and *stmt2* concurrently.

We illustrate Esterel's syntax and semantics by means of a small example; note that this example is not intended to have any particular meaning.

```
output X : integer;
input A1, A2;
input B : boolean;

module test_program:
  emit X(10);
  loop
    await A1 || await A2;
    await immediate B;
    pause;
    if ?B then
      present A1 then
        emit X(100);
      end
    end
  end
end module
```

Our example program has one output signal `X` carrying an integer value, two input signals `A1` and `A2` carrying no values, and one input signal `B` carrying a Boolean value. As can be seen, Esterel programs are split into modules, so as to support the concepts of program decomposition and software reuse.

In the first instant, i.e., reaction cycle, of our example program, value `10` is emitted on output signal `X`, and the `loop` body is entered. This is followed by the `await A1 || await A2` statement which tells the program to concurrently wait for input signals `A1` and `A2`, and ends the first instant. The program now continues waiting for `A1` and `A2` forever, so let us assume that these signals are received from the environment sometime before the second instant. Therefore, in the second instant, statement `await A1 || await A2` completes and `await immediate B` is executed. The use of the `immediate` keyword means that the input signal can arrive in the instant that executed the `await immediate` statement, i.e., it does not force the next instant to take place like statement `await`. Instead, a `pause` statement can be used in oder to force the next instant to occur. The `present` statement shows how to access values carried via signals, namely through the use of the `?` operator. Hence, if the value of signal `B` read in the previous instant is true and if input signal `A1` is present, then the integer 100 is output via signal `X`, before the loop executes again.

The semantics of Esterel is well defined and has been extensively investigated in the literature [4]. In addition to the *synchrony hypothesis* which underlies th concept of cycle–based reaction, the semantics is based on the principles of consistency, causality, reactivity

and determinism. *Consistency* means that a signal cannot be both present and absent within the same instant. Every presence and absence of a signal must further be *causally* justified, by ultimately referring to the presence and absence of the input signals. Moreover, a program must permit a reaction, no matter what the statuses and values of the input signals are, i.e., it must be *reactive*. It must also compute a unique reaction in each instant, for each possible input, whence it is *deterministic*.

It has been mathematically verified that Esterel's semantics possesses many desirable properties. In particular, the set of all valid Esterel programs corresponds one–to–one to the set of those asynchronous digital circuits with feedback that stabilise independently of any gate or wire delays [23]. This close relationship between programs and circuits is utilised by the code generators available in Esterel Studio. While creating VHDL or Verilog from a valid Esterel program involves synthesising the circuit corresponding to the program, Esterel Studio's C code generator essentially simulates this synthesised circuit.

Last, but not least, it must be pointed out that Esterel is an extensible language, which allows users to define external data types, external functions and procedures, as well as tasks. Given that the processing of reactions must be quicker than the system's environment, it seems reasonable to disallow any operations to take place during the kernel processing that might cause delay to a subsequent instant. It is for this reason that external functions and procedures in Esterel must be instantaneous [3]. If asynchronous execution of external code is desired, then tasks ought to be used. We have made extensive use of Esterel's extensibility features during the development of our database APIs.

3 Database APIs for Esterel

In this section we devise two different APIs for enabling relational database access within Esterel, depending on whether result sets to queries are stored locally or remotely to the Esterel reactive kernel. Both allow multiple, simultaneous connections to databases and are intended for use in different application scenarios.

We start off by providing the rationale for developing *two* APIs, for which we review the possible computing architectures running an Esterel reactive kernel and a database. Both our APIs consider databases as part of the system environment and as running asynchronously to the reactive kernel. This is because database transactions are typically more complex to process than ordinary reactions. A typical database interaction consists of several stages. First, the database is queried and a *result set* is generated according to that query. Such a result set is simply a set of rows, possibly ordered, that contains the data specified in the query. From this point on, most database management systems allow for two different routes to access the result set. One method is to transfer the whole result set to the client that issued the query, and let the client use the information as necessary. The alternative method is for use in situations where it is infeasible to transfer the whole result set, due to memory or bandwidth constraints. Instead, the client machine may access the result set stored on the database server in a row–by–row manner, minimising the speed at which data needs to be transferred to the client machine. This dual–retrieval method offered for the result set is what facilitates our decision to develop two APIs instead of a single one:

The *Local Result Set API* views the result set as being local to the reactive kernel. In this case, operations on the result set can conceptually be considered to be instantaneous, thus satisfying the synchrony hypothesis [12]. This API will therefore make heavy use of user–defined external functions and procedures, which require synchrony. In practise, the Local Result Set API is for use when there is either a high–speed link between the database server and the reactive client, or when both reside on the same physical machine.

The *Remote Result Set API* considers the result set to be stored remotely to the reactive client, i.e., the result set — in addition to the database — is viewed as part of the reactive system's environment. This allows for the result set to be transferred row–by–row to the client, as is standard practice when accessing relational databases and closely fits with the provisions of MySQL. This is for use when, e.g., the link between the database result set and the client is much slower than the duration of a reactive cycle. As a consequence, transfers are not instantaneous, and the Remote Result Set API cannot employ Esterel's elegant mechanism of external functions and procedures, but must rely on *signals* and *tasks* instead. Indeed, a task concept has been incorporated into Esterel exactly for the purpose of handling external, asynchronous computations.

In the following we discuss the design and usage of both APIs, first the Local Result Set API and then the Remote Result Set API. Some of the implementation details of the APIs will be presented later in Sec. 4; in particular, users will be able to customise the APIs according to their wishes, e.g., in terms of the number of database connections required and the frequency with which they are accessed.

3.1 Local Result Set API

The easiest way to view the interaction between a reactive kernel programmed in Esterel and a database is to regard the database simply as an extension of the reactive kernel's environment. For this reason all interactions with the database from within Esterel are modelled using input and output signals, as these are Esterel's facilities for communicating with the environment. Therefore, to perform an operation on the database, a dedicated output signal is emitted, parameterised in a string that formulates a query in SQL syntax. The database's response is awaited via a dedicated input signal whose parameter carries an identifier that points to the result set. During the time between the emitted query and the results returning, the database is queried and the whole result set is transferred back to the site that also runs the reactive kernel. Note that multiple databases can simply be supported by declaring a dedicated output and input signal for each database.

Once a database has been queried and a result set returned, data can be extracted from the result set via dedicated operations, for which we employ Esterel's external function and external procedure facilities. This is possible since both the result set and the reactive kernel reside in the same memory, which implies that accesses of the result set by the kernel may be considered as instantaneous. If the query's SQL command is one that does not return results, such as the command for the deletion of data items, then the only operation provided is one to check the number of affected rows. If the SQL command did return a result set, however, the set may be accessed by successively reading it row–by–row, extracting the specific data items from each row and coercing them into native Esterel data types. Once all rows have been processed, an operation shall be called to free the memory occupied by the result set. Note

Table 1: Services offered by the Local Result Set API

```
type MYSQL_RES_ptr;
type MYSQL_ROW;

procedure appstr() (string, string);
procedure appint() (string, integer);
procedure appbol() (string, boolean);
procedure appflt() (string, float);
procedure appdou() (string, double);

output <Signal name for emitting query> : string;
input <Signal name for returning results> : MYSQL_RES_ptr;

function check_result(MYSQL_RES_ptr) : boolean;
function get_next_row(MYSQL_RES_ptr) : MYSQL_ROW;
function num_rows(MYSQL_RES_ptr) : integer;

function getstr(MYSQL_ROW, integer) : string;
function getint(MYSQL_ROW, integer) : integer;
function getbol(MYSQL_ROW, integer) : boolean;
function getflt(MYSQL_ROW, integer) : float;
function getdou(MYSQL_ROW, integer) : double;

function num_affected_rows(MYSQL_RES_ptr) : integer;
procedure clear_results() (MYSQL_RES_ptr);
```

that the three main operations (querying, retrieving results and clearing results) must always be conducted in this order; for example, emitting a second query, before having received and cleared the results for the first one, results in undefined behaviour. This is also true analogously for the Remote Result Set API. Esterel's interaction with a remote database and its local processing of result sets thus leads to the API displayed in Table 1. In the remainder of this section we explain and illustrate the API's services in more detail.

We begin with the formation of a query string containing SQL commands. Since Esterel does not provide any facilities for building strings, the string must be generated using a series of *append* operations. The API offers an append operation for each of Esterel's native data types and implements these operations in Esterel using external procedures. As an example, the following Esterel program fragment generates a query related to our warehouse case study, which uses an integer variable `order_id`:

```
var query_str : string in
    query_str := "select * from orders where order_id = ";
    call appint() (query_str, order_id);
end var
```

As mentioned before, one interacts with the database via an output query signal and an input result signal. For each database used, these signals should be declared as such:

```
output item_db_query : string;
input item_db_results : MYSQL_RES_ptr;
```

Each pair of signal names can be selected by the user. The mapping between these signal names and the actual databases is defined elsewhere and will be explained in Sec. 4. The data returned on a result signal is simply an identifier of the external type `MYSQL_RES_ptr` which is defined in our API's implementation. Our framework effectively allows only one result set per database connection; however, if more result sets are required simultaneously within some Esterel application, additional connections to the same database may be declared.

In order to ensure that the results are received correctly from the database, the result signal should be *awaited* right after the emission of the query:

```
emit item_db_query("select * from item");
await item_db_results;
```

If two queries are issued simultaneously, then the result signals must be awaited in parallel or using *immediate await* statements in Esterel. This is due to the return signal from a database query only being present for one cycle; the program must register the signal on the cycle it is present or it will be lost. Therefore, if the results of two queries are being awaited, the program must be able to recognise both on the same cycle. For example, suppose we have a second database connection to the one shown above, with signal names `order_db_query` for the query signal and `order_db_results` for the signal returning the results. If we require a query on both connections to be issued simultaneously, then this should be done as follows:

```
emit item_db_query("select * from item");
emit order_db_query("select * from order");
await item_db_results || await order_db_results;
```

To check the success of the SQL command, the Boolean function `check_result` should be called and passed the identifier of the result set, i.e., the value of the input result signal. If it returns true, then the query has succeeded and the operations described below may be used to access the data inside the result set. If it returns false, then the data in the result set is not valid. When querying a database, the query may fail in a number of ways, ranging from a timeout to an incorrectly formed SQL statement. The `check_result` function is suitably abstract so as to account for these problems, and simply lets the Esterel system know the outcome of performing the query. It is the responsibility of the programmer to check the success of a query, and any failure modes should be specified in Esterel. An example of how this may be done using the *trap* and *exit* statements of Esterel's error handling mechanism is given in the case study in Sec. 5.

For working with a result set that contains data — as opposed to an empty one returned by, e.g., an SQL insert statement — rows must be declared inside Esterel. Rows are declared to be of external type `MYSQL_ROW` that is defined in our API's implementation. The lifetime of any data loaded into a row from a result set lasts only as long as the result set itself, i.e., up to the time the `clear_results` operation is called. Therefore, rows should only be declared locally and in such a way that their scope finishes before the call to `clear_results` occurs.

The functions provided to operate on the result set and rows will now be described. Most of the operations mirror the equivalent MySQL functions from the MySQL C API [20] on which our implementation is based. This is because the MySQL C functions are widely known. Moreover, at a later date, additional functions can easily be included, if desired. Function `get_next_row` is required to load data into a row from a result set. It is passed a result set identifier and, each time it is called, it will return the next row in the result set. Generally, the program will need to know how many rows there are in the result set and, therefore, how many times to call `get_next_row`. This is accomplished by a call to function `num_rows` which, when passed a result set identifier, returns the number of rows in the result set. Once a row has been loaded from a result set, data can be extracted using a `get<type>` function which is provided for each of the native Esterel data types. In addition to a row, an integer is also passed and indicates the index of the column from where the data is to be retrieved. The following is a simple example of data extraction using our API:

```
var row_holder : MYSQL_ROW,
    item_name : string,
    item_location : integer in
  row_holder := get_next_row(?item_db_results);
  item_name := getstr(row_holder, 1);
  item_location := getint(row_holder, 2);
end var;
```

In this case, the results are identified by the valued signal `item_db_results`, and the item's name and location are stored in the second and third column of a row, respectively. Note that the indexing of columns is adopted from the C language and thus starts with 0.

Function `num_affected_rows` for accessing the result set is used when the result set contains no data but a user wants to know how many rows were affected by the SQL command. As such, `num_affected_rows` can be employed to test the success of an SQL query, e.g., to check whether a delete query has had the desired outcome.

The final operation provided by the API, to which we have already referred above, clears the memory occupied by the result set:

```
procedure clear_results()(MYSQL_RES_ptr);
call clear_results()(?item_db_results);
```

It is essential that there are no rows loaded from the result set after it is cleared, since the data within these rows is cleared with the result set as well.

3.2 Remote Result Set API

The Remote Result Set API should be used in situations where it is not feasible to transfer the entire result set to the system running the reactive kernel, i.e., when both the database *and* the result set must be viewed as part of the environment. Since remote communication must be taken into account, the API is different to the Local Result Set API. This is because external functions and procedures can only be used in Esterel if their operations may be considered instantaneous [3]. Consequently, one must either employ Esterel's task concept or

must solely rely on signals for the Remote Result Set API. In both cases and as a consequence of operations on the result set not being instantaneous, the Esterel kernel must be informed when an operation is complete. This is accomplished by awaiting an "acknowledge" task completion signal after every operation. In the remainder we focus on our solution via tasks rather than signals, since this is the most elegant method for representing asynchronous interaction in Esterel. The interested reader is referred to [27] for an exposition of the solution employing signals.

An important aim of the Remote Result Set API is to minimise the amount of data that must be transfered, due to memory or bandwidth being at a premium. For this reason, rows are transfered to the system running the reactive kernel one at a time. There should be no overhead in returning the whole row, as opposed to the individual elements, since the row structure is specified in the query, and therefore it is the programmer's decision exactly of what data the row consists. To prevent the complexity of handling multiple rows in the kernel, each database connection is limited to passing only one row at a time. This is a reasonable restriction since systems that use this API are unlikely to be performing complex database manipulations that require multiple rows.

Table 2: Services offered by the Remote Result Set API

```
procedure appstr() (string, string);
procedure appint() (string, integer);
procedure appbol() (string, boolean);
procedure appflt() (string, float);
procedure appdou() (string, double);

task <db_id>_perform_query () (string);
return <db_id>_perform_query_complete0(boolean);

task <db_id>_get_row () ();
return <db_id>_get_row_complete0(boolean);

task <db_id>_clear () ();
return <db_id>_clear_complete0;

function <db_id>_getint(integer) : integer;
function <db_id>_getstr(integer) : string;
function <db_id>_getbol(integer) : boolean;
function <db_id>_getdou(integer) : double;
function <db_id>_getflt(integer) : float;
```

Our API for remote result set access is displayed in Table 2. The remainder of this section explains the API's services. Similar to the naming of the query and result signals in the Local Result Set API, each task `task_name` and function `function_name` is prefixed with a textual database identifier `db_id`, which we denote by `<db_id>_task_name` and `<db_id>_function_name`, respectively. Due to the restrictions imposed on tasks by Esterel, for each occurrence of starting a task in the syntax, a unique return signal is required to inform the reactive kernel of task completion. We represent this in the API by appending

each return signal name, e.g., `<db_id>_query_complete`, with a unique number. For implementation reasons, this should start at 0 and increment by 1 for each task completion signal required:

```
task <db_id>_perform_query () (string);
return <db_id>_perform_query_complete0(boolean);
return <db_id>_perform_query_complete1(boolean);
return <db_id>_perform_query_complete2(boolean);
...
```

Note that the offered string generation functions are identical to those in the Local Result Set API, as described in Sec. 3.1.

The main difference to the Local Result Set API is the way in which the results to a query are accessed. There is now one result set and one row per database defined, and since each signal is prefixed by a unique string, there is no need for a result set identifier to be returned. The only data returned after issuing a query is the success of that query. Therefore, after a query has been issued, the Boolean return signal for that task must be awaited syntactically just after starting the task:

```
exec <db_id>_perform_query()("select * from item")
  return <db_id>_perform_query_complete0;
await <db_id>_perform_query_complete0;
```

Note that the signal names `<db_id>_perform_query_complete0` in the `return` and `await` statements must match; they are a pair. It is an obligation on the programmer to ensure that there is no mismatch. This particularity cannot be resolved within the API as it is limited by Esterel's restrictions on task programming.

The value carried by the return signal is the same as that returned by the `check_result` operation in the Local Result Set API, i.e., it should then be tested to determine whether the query has succeeded or not. If the query has succeeded and the result set is not empty, then the first row can be transfered by executing the `<db_id>_get_row` task. The Boolean return signal carries the value *true* if there exists a valid row to load, and *false* if there are no more rows available. Now that a row has been loaded, its elements can be accessed in a manner identical to that of the Local Result Set API, except that the database prefix `<db_id>` is used instead, since no row identifiers exist in this Remote Result Set API.

The operations for determining the number of affected rows and the number of rows in the result set are not supported by MySQL when the result set is stored server side. Therefore, to step through the rows in a result set, the value of the return signal from the `get_row` task must be used as the loop variable. When there are no more rows left, it will carry value *false*.

3.3 Trade–offs between the APIs

While we have already pointed out the different situations for which the Local and Remote Result Set APIs have been designed, this section discusses the trade–offs between the APIs.

The Local Result Set API is based around flexibility and ease of use, which are a consequence of the fact that it is built upon external functions. In contrast, the Remote Result Set API is not as elegant by comparison, as it requires, e.g., an "operation complete" signal to indicate the success of a database operation, whereas in the Local Result Set API this is given as the return value of an external function. Furthermore, the Esterel language specifies that, for each syntactic occurrence of starting a task, its completion must be awaited by a unique signal. Therefore, if an Esterel program includes a large number of occurrences of querying a particular database, then the number of return or task completion signals will be equally large. It should be noted that the querying of the database in the Local Result Set API is asynchronous, just as in the Remote Result Set API. However, in the former we do not use tasks to represent this, but instead rely on signals to illustrate the differences between the two methods and to emphasise that the database is considered part of the environment. As is the case regarding ease of use, the Local Result Set API also compares favourably to the Remote Result Set API regarding performance; this is because of the use of external functions rather than tasks.

Another difference between the two APIs is the number of rows that can be used simultaneously. The Local Result Set API supports as many rows as can be stored in memory, whereas the Remote Result Set API is limited to accessing one row at a time. However, since the Remote Result Set API is meant for use on small embedded systems with slow communication links, it is not expected that database operations requiring many rows will be commonplace.

Both APIs support connections to multiple databases. In the case of the Remote Result Set API, this can be used to overcome the limitation of one row per database by simply defining a second connection to the same database. Since database connections are generally permanent throughout the time a reactive system is running, our APIs provide no explicit facilities for connecting and disconnecting from a database. Instead, connection and disconnection is handled implicitly by the implementation of the APIs (see Sec. 4).

Finally, we discuss the issue of timeouts for when database transactions over–run. Timeouts must be present on all database transactions or the Esterel system could be left waiting for a completion signal that will never arrive. Currently, timeouts are simply coded within our APIs and assumes that each database transaction uses the same timeout value. If variable timeout values should be desired, one could adapt our APIs such that these timeouts could be encoded directly in Esterel; we leave this for future work. However, if a *real–time database* [16] with guaranteed response times is available, then timeouts would no longer be an issue. In certain application scenarios, it may even be possible that a real–time database could offer such good response times that the database would no longer need to be considered as part of the environment and could be directly integrated into the reactive system allowing the use of external functions and procedures for querying.

4 Implementation of the APIs

This section gives some details on our implementation of the Local and Remote Result Set APIs; the full source code is freely available for download from [26].

Both APIs are implemented in a combination of Perl and C and rely on the MySQL C API [20]. The involvement of the Perl scripting language [25] in the realisation of the APIs may be surprising at first. The reason is our desire to support *multiple* databases with *user–declared* signal names for emitting SQL queries and awaiting result sets. As reactive systems typically interact with an arbitrary but fixed number of databases, it is unnecessary to provide API services for dynamically binding signal names to databases. Even a static binding should not be defined within an Esterel program at all, as it is not part of specifying reactive behaviour. Instead, we choose to provide such a binding as a parameter to our Perl script for each API, which appropriately combines the C code generated by the Esterel compiler [10, 22] with C code implementing the API services used in the underlying Esterel program. Perl is an ideal choice to build this combination of C code and API services due to its fast and simple operators for writing and reading to text files. Furthermore, Perl's regular expression feature is very useful for locating code segments that must be copied out of the Esterel generated C code in a robust manner.

The implementation of the Local Result Set API will be given preference here. This is because it is similar in spirit to the Remote Result Set API implementation, but does not require the added complication of threads and deeper knowledge of POSIX [8] as outlined in Sec. 4.2. The structure of an Esterel program once it has been translated into C, is that of an automaton which should be called on each instant of the system. Prior to being called, the input signals should be set up. The automaton will in turn call any output signals that are emitted on that instant. The implementation of the Local Result Set API is therefore based around the central automaton call. When a query is sent to the database, the automaton calls a user–written procedure that records its presence. Similarly, before the automaton is called, input signals are set up to reflect any response from the database.

4.1 Detailed Implementation of the Local Result Set API

The local result set API script is used as follows:

```
gendb.pl <Main Module Name>
         <Max length of strings and queries>
         <DB Name>
         <Host>
         <User>
         <Password>
         <Signal name for emitting query>
         <Signal name for returning results>
```

The parameters carry the following meaning:

- *Main Module Name:*

  The name of the module compiled with Esterel; it is assumed that the auto–generated C code is in `<Main Module Name>.c`.

- *Max length of strings and queries:*

  Since all strings must have a fixed–length representation in Esterel, this specifies what the maximum length is. It must be ensured that the program never exceeds it. This includes getting strings from the database using `getstr()`. The type of data that `getstr()` is called on should be guaranteed not to exceed the maximum length; e.g., if the maximum length was 200, the database field could be of type `VARCHAR{150}`, and the string could be appended by up to 50 characters by the user to represent the SQL query.

- *DB Name:* The name of the database on the host.

- *Host:* The location of the MySQL DB; if hosted locally, then `localhost` is to be used.

- *User:* The username to be used to connect to the database.

- *Pass:* The associated password.

- *Signal name for emitting query:* The name of the signal that is used to send a query to this database, i.e., the name of the signal listed as

  ```
  output <Signal name for emitting query> : string;
  ```

  in Table 1.

- *Signal name for returning results:* The name of the signal that will be awaited for results, i.e., the name of the signal defined as

  ```
  input <Signal name for returning results> : MYSQL_RES_ptr;
  ```

  in Table 1.

The last six parameters may be repeated any number of times for additional databases, whence a single database can be accessed via multiple connections.

Note that we have intentionally not provided default values for the parameters of the `gendb` script. This is because we believe that every application scenario will be different and that there are no sensible default values.

After the arguments to the script have been processed, it continues by copying the relevant parts of the Esterel generated C code into the new C code file. For example the following Perl code copies consecutive lines of `EST_FILE` to the file `OUT` until the line `#include "${main_module}.h"` is found. During this process, when the line defining the maximum string length is found, it is replaced with the user–defined string length:

```
while (($line = <EST_FILE>) ne "#include \"${main_module}.h\"\n") {
    if ($line =~ /#define STRLEN/) {
         print OUT ("#define STRLEN ${max_strlen}\n"); }
    else {
         print OUT $line; }
}
```

Perl's treatment of file handles and regular expressions makes this kind of code manipulation very easy. Note the command `$line = <EST_FILE>` which reads the next line of `EST_FILE` into the variable `$line`, and the regular expression operator `=~` which tests if a string variable contains a regular expression. If the symbol " occurs inside a sting it must be escaped using \. Furthermore, variables can be directly included in strings. This occurs in the string `"#include \"${main_module}.h\"\n"`, where the string value in the variable `$main_module` is substituted wherever `${main_module}` appears.

The next operation is to define all MySQL variables; this is based on the number of database connections required by the user. In the code below, the array `dbs[$x]` stores all of the user–entered information for the database connection `$x`. The entry `$dbs[$x][0]` is the id for that connection:

```
for (my $x=0; $x<$num_dbs; $x++) {
  print OUT ("\n//Global variables for ${dbs[$x][0]}\n");
  print OUT ("MYSQL init_${dbs[$x][0]}, *sock_${dbs[$x][0]};\n");
  print OUT ("MYSQL_RES *result_${dbs[$x][0]};\n");
  print OUT ("int query_pending_${dbs[$x][0]},
              query_suceeded_${dbs[$x][0]};\n");
}
```

After this, the functions and procedures defined in Table 1 are produced. For example, the Esterel function `get_next_row(MYSQL_RES_ptr) : MYSQL_ROW;` is produced like this:

```
MYSQL_ROW get_next_row(MYSQL_RES_ptr res) {
  return mysql_fetch_row(res);
}
```

Next, the signal output function is produced for each database query signal. This function is responsible for submitting the query to the database, and for retrieving the result or reporting a failure. The code is too long to include here, but the interested reader is referred to view it in the source code of the Perl script where the C functions used to interface with MySQL are clearly seen [26]. Finally, the main C procedure is produced. It commences by initialising all database variables and the automaton state. If a connection to a required database cannot be opened, execution is terminated. The main reactive loop is started, inputs are set up, and the main automaton step procedure is called.

Sec. 3 mentions that databases are running asynchronously to reactive kernels, since database transactions are relatively complex and thus cannot be assumed to respect the synchrony hypothesis. The current implementations of our Local Result Set API does not implement this intention explicitly. This is because the output procedure called by the reactive kernel to process a database interaction should be able to be considered instantaneous. However, due to the fact that we wait for the results from the database in the same procedure, this will not be the case. This is however not a problem in the API since, immediately following a query emission, the result input signal should be awaited. Since there can be no statement between these two operations, it does not matter that the reactive cycle has temporally paused. Timeouts are used to make sure the database query output procedure cannot execute forever, in case of a problem with the connection to the database.

However, if an implementation of our API should explicitly support the asynchronous view between synchronous reactive kernels and databases, then it would not be difficult to do so by introducing threading using POSIX [8]. The query output function would need to be changed to inform a separate thread that would perform the database operation at hand. At the start of each iteration of the reactive system, the thread would then be queried to determine if any database operations have finished. If so, the location of the result set would be passed to the reactive kernel thread, and the database result input signal would be emitted in that cycle.

4.2 Discussion of the Implementation of the Remote Result Set API

In the implementation of the Remote Result Set API, the actual database transactions are handled in a separate thread to ensure that they do not interfere with the periodic nature of the automaton. One thread is made for each database connection. Its task is to regularly check if the automaton has requested it to perform a database operation and, if so, it accesses the associated parameters and performs the operation. When the operation is complete, its results are reported back through a shared-data store and it waits for the next database operation.

The passing of information between the thread running the main automaton and a thread running a database connection must be carefully controlled in order to ensure the automaton thread does not become stalled. Broadly speaking, this means imposing restrictions on when data can be sent to a database thread. In terms of our API, this equates to the restriction that only one database operation can be performed at a time, i.e., the operation must be fully complete before the next is begun. To explain the interaction between the two types of threads, we will now give a code skeleton of each thread, followed by an example of how a typical database operation is accomplished.

```
MUTEX: Mutex1

THREAD: Database Connection 1 (c)
loop forever
  if (acquire lock on Mutex1)
    if (database operation to perform)
      Perform request;
      Record results;
    end;
  Release lock on Mutex1;
  end;
  Yield runtime;
end

THREAD: Esterel Automaton
loop forever

  // Setup inputs (d)
  if (acquire lock on Mutex1)
```

```
    if (database operation to perform)
      if (database operation completed)
        Setup up the automaton inputs with the database operation result;
      end
    end;
    Release lock on Mutex1;
  end;
  Perform Main Automaton Call; (a)

  // Process outputs
  For any new task that was started in the last cycle
    Call the output function associated with that task (e.g., Start Task 1);
end

FUNCTION: Start Task 1 (b)
wait until (acquire lock on Mutex1)
  Record that THREAD: Database connection 1 must perform a db operation;
  Release lock on Mutex1;
end
```

A typical database operation will now be described to illustrate where blocking can occur and how it is handled. Let us begin by assuming that we are currently executing the automaton step procedure (a) and that a task requiring a database operation has just been started. When the automaton step procedure ends, we will execute the output function (b) associated with the task that was just started. This function will wait until it has acquired a lock on Mutex1. Once the lock is acquired, it will record that the database operation thread (c) must perform an operation when it is next scheduled. It then releases the lock and returns.

This is the only part of the process where blocking can occur as it is possible that the database connection thread has the lock on Mutex1. However, since our API only allows one operation to be executed at a time, the database connection thread cannot be performing a time consuming operation, but instead must be checking to see if there is an operation to execute. This check is very fast, and the lock will be released and its runtime yielded quickly. This will allow the output function (b) to acquire the lock and complete.

The next time the database connection thread is scheduled after the output function has completed, it will try to acquire the lock on Mutex1 and find that there is an operation to execute. It will then execute this operation, record the results, release the lock and yield its runtime. The next time the setup inputs phase (d) of the main reactive loop is executed (i.e., after the database connection thread has finished its operation), it will acquire the lock on Mutex1 and setup the automaton inputs with the results of the database operation. After this, it will release the lock on Mutex1, and the API's side of the operation is complete.

If the acquisition of the lock fails during the input setup phase (d), then the database operation thread is busy and no results need to be passed back to the automaton; therefore, the input setup phase for database results can be skipped in this iteration. In case the database connection is unreliable, Mutex1 could become locked forever; timeouts are included to prevent this. The full implementation details of the Remote Result Set API can be found in [26].

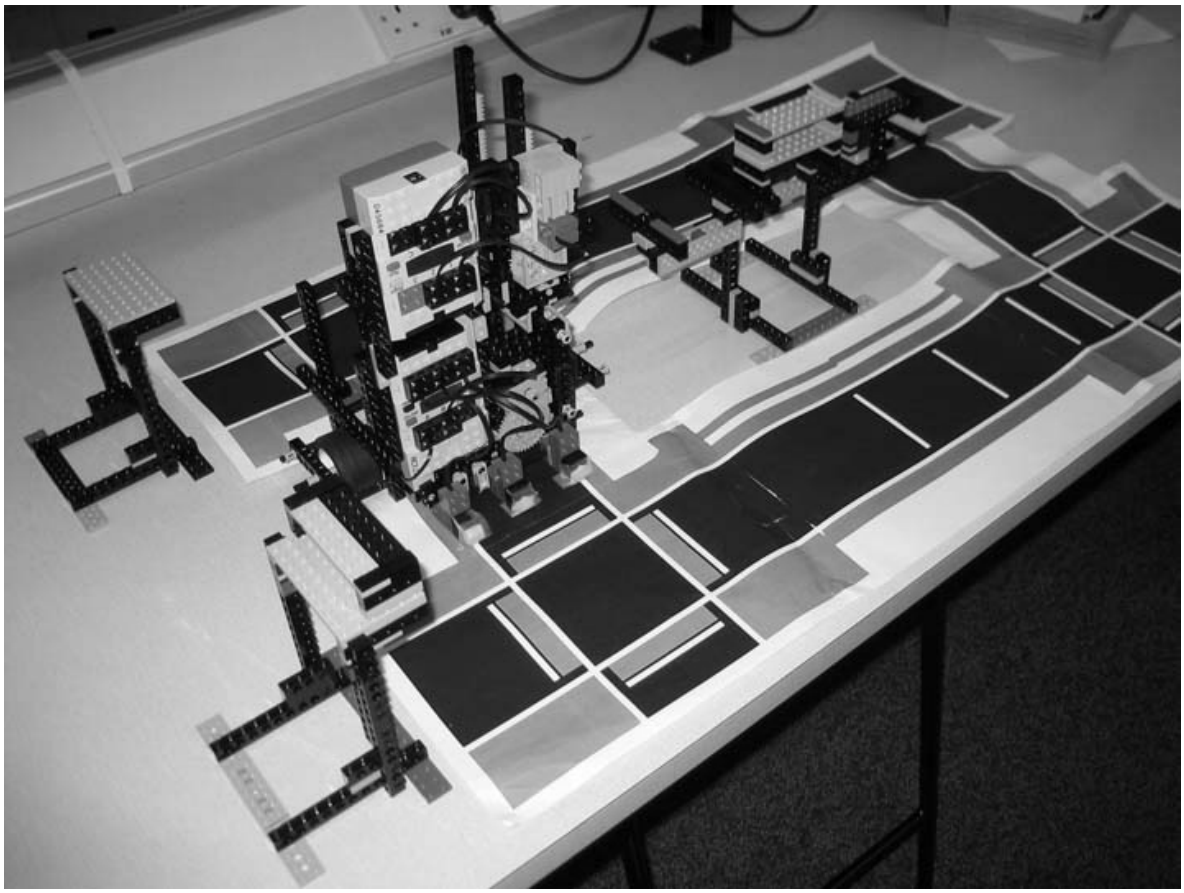

Figure 1: Aerial photograph of our warehouse system constructed using Lego Mindstorms.

5 Case study

In this section we present a case study demonstrating the utility of our APIs: an automated warehouse storage system modelling a direct order company, where items from orders are picked, stored and finally removed from the warehouse. This involves producing a warehouse containing various items and a robot capable of moving the items within the warehouse, for which we use *Lego Mindstorms* robotics kits [17] (cf. Fig. 1). Lego Mindstorms provides both a construction tool with sensors and actuators and a micro–controller, called the RCX, which is capable of running a reactive kernel programmed in Esterel. The database behind our warehouse model is that of a standard order system but which also includes mapping data about the location of the items. As this case study is meant to exemplify the use of our database APIs, only the part of the solution employing the APIs is emphasised below. All code and associated scripts for the case study may be downloaded from [26], where the interested reader can also see a video of our warehouse system in operation.

5.1 Lego Mindstorms, the RCX and BrickOS

Lego Mindstorms is a platform for building computer–controlled robots using the *Lego system* [17]. At the heart of Lego Mindstorms is the RCX. This "brick" is a small battery–powered computer which is capable of controlling up to three *actuators* and reading up to three *sensors*. In Lego, actuators are normally motors, and sensors can be light, rotation and touch sensors. Each RCX also provides an *infrared transmitter and receiver* used for both downloading programs from a PC and for inter–RCX communication. The infrared download device used on the PC can also participate in communications with RCXs.

The RCX provides great flexibility through its re–programmable firmware. BrickOS [7], formerly known as LegOS, is an open–source replacement firmware for the RCX. It boasts a number of features that make it considerably more powerful than the standard Lego firmware. Foremost, it allows programs written in the C language to be executed on the RCX. Obviously, this is especially important for this project since the Esterel compiler [10] generates C code as target language. BrickOS also provides infrared communication through the *Lego Network Protocol* (LNP) [7] which allows message broadcast and directed transmissions.

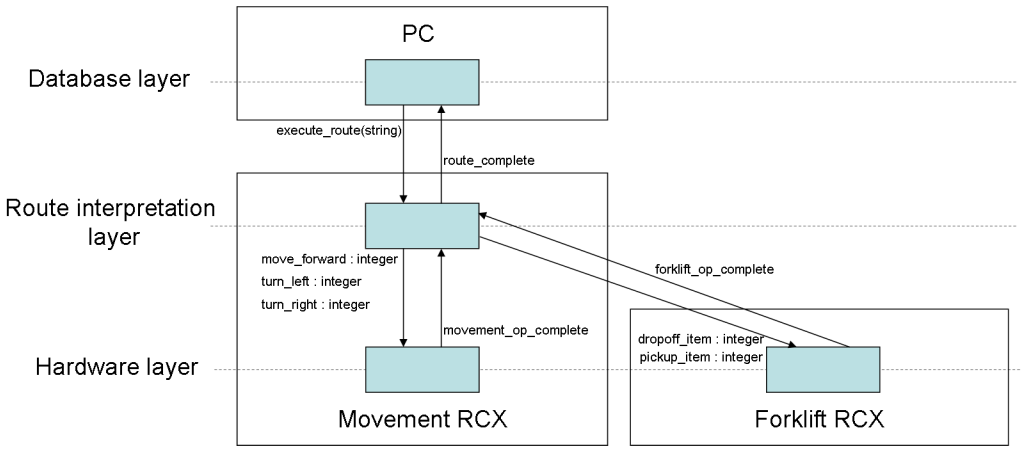

Figure 2: Diagram showing the inter–layer signals (arrows) and physical components (transparent boxes). Communication between physical components is via an infrared link.

5.2 Hardware

The hardware requirements of our warehouse storage system are high in Lego Mindstorms' terms, requiring more sensors and actuators than one RCX can control. Therefore, it is necessary to use two RCXs, one to control the movement of the robot and the second to control the employed forklift installed on top of the movement unit, hereafter referred to as the *Movement RCX* and *Forklift RCX*, respectively. Again, communication between the two RCXs is handled using the infrared link provided on each RCX. Because the Forklift RCX does not need to communicate with the PC running the database system, the infrared download tower is set up to allow the following communication to take place: PC to Movement RCX and Movement RCX to Forklift RCX, as shown in Fig. 2.

5.3 Software

The software for the warehouse system is structured in three layers: *database access layer*, *route interpretation layer* and *hardware layer*. Each device used in the system, the two RCXs and the PC, executes a reactive kernel programmed in Esterel; note that these are running asynchronously to each other, due to the non-negligible delay caused by infrared communication channels. The database layer runs on the PC and is responsible for accessing the warehouse's database and for generating routes through the warehouse for the robot to execute. The second layer runs on the Movement RCX and interprets the route sent from the PC. The third layer is responsible for interacting with the Lego hardware and performs operations such as *move the robot* and *pick up the item*. This layer is present on the Movement RCX and the Forklift RCX. The signals used for communicating between the various layers are shown in Fig. 2.

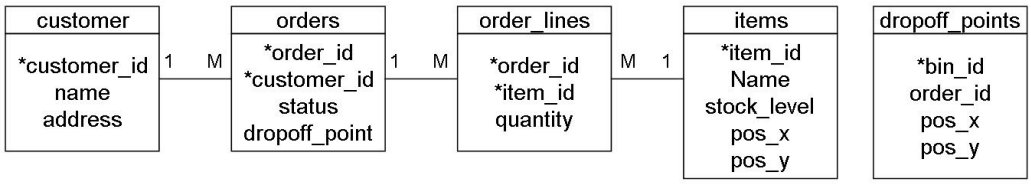

Figure 3: The employed database schema.

5.3.1 Database

Our warehouse's database models a simple ordering system. Since the main emphasis within this case study is on retrieving spatial data, the aspects of the database concerning ordering information are kept as simple as possible: a customer may make multiple orders, each order is identified by an order id and must contain one or more order lines, and each order line refers to exactly one item. The corresponding database schema is given in Fig. 3.

Stored separately is a table describing the drop–off bins in the warehouse. A drop–off bin is used by the robot to place parts of an order before it is complete. When the order is complete, the items that are contained in the bin are removed, and the bin is then ready for the next order. For each bin, its location is stored, as is the id of the order in the bin, in case the bin is in use.

5.3.2 Database access layer

The database access layer uses our Local Result Set API. We chose the Local Result Set API for our case study since the Esterel program interacting with the database will be running on the same machine as the MySQL server. Therefore, result sets will be stored in the same memory as the Esterel program is run in, and can be considered to be accessible instantaneously by the reactive kernel. Two connections to the same database are maintained since, at one point in the program, it is necessary to manipulate the database while retaining the result set of an earlier operation. The input and output signals for the main connection to

the database are called `orders_query_out` and `orders_results`, respectively, and for the additional connection `stock_query_out` and `stock_results`, respectively, since those are only used to update stock levels:

```
output orders_query_out : string;
input orders_results : MYSQL_RES_ptr;

output stock_query_out : string;
input stock_results : MYSQL_RES_ptr;
```

Since the database access layer's only function is to wait for an order that needs to be picked and then to instruct the robot how to pick it, the main module is constructed as a loop. Inside the loop is a `trap` statement which handles all the ways in which the database and the system can fail (see below). Therefore, all failure modes are dealt with in one place, and the error handling process is simplified.

The operation of the database access layer is roughly as follows. First, an order is retrieved from the database. One of the lines of this order is then extracted and the item details are stored locally. The robot is then sent to retrieve all items, one by one, and to deliver them in an available drop–off bin using a pre–generated route through the warehouse. After each item has been collected, it is necessary to update the stock level of the item's type. However, at this point the database result set still contains uncollected order lines which are required for later in the program's execution. Therefore, the second database connection is used to perform the update without overwriting the result set containing the order's details.

Since the functionality of the database access layer is quite simple and most database operations are effectively the same, only two database operations will be discussed. The retrieval operation starts by emitting the query on output signal `orders_query_out` and then awaits the results:

```
emit orders_query_out("select order_id, customer.customer_id,
    name, address from customer, orders where
    customer.customer_id = orders.customer_id and
    orders.status = 'AWAITING_PICKING' order by order_id");
await orders_results;
```

The returned results are then checked for validity using our API function `check_result`. If the query has succeeded, then `num_rows` is called on the result set to see if any rows were returned. If rows have been returned, then there are orders waiting and, consequently, the first row is loaded into the local variable `row`. The details of the row are then retrieved using the `get<type>` functions and emitted on the corresponding local signals: `order_id`, `customer_id`, `customer_name` and `customer_address`. Now that the result set is no longer needed, the `clear_results` procedure is called. The following code captures these steps:

```
if (check_result(?orders_results)) then
  if (num_rows(?orders_results) > 0) then
    var row : MYSQL_ROW in
      row := get_next_row(?orders_results);
```

```
      emit order_id(getint(row,0));
      emit customer_id(getint(row,1));
      emit customer_name(getstr(row,2));
      emit customer_address(getstr(row,3));
    end var;
    call clear_results()(?orders_results);
  else ...
```

Note that we only look at the first order and then clear the result since we simply wish to process one order at a time. Once the `order_id` has been obtained, we are not interested in the result set any more.

There are two ways in which this operation for retrieving orders can fail: firstly, if the query fails and, secondly, if there are no waiting orders. Each is catered for by an `exit` statement which corresponds to the `trap` mentioned above:

```
if (check_result(?orders_results)) then
  if (num_rows(?orders_results) > 0) then
    %Code snipped
  else
    call clear_results()(?orders_results);
    exit no_waiting_orders;
  end if;
else
  exit bad_query;
end if;
```

In each case, the program flow jumps to the end of the loop and emits an appropriate error message before pausing and then repeating the main loop. Note that function `clear_results` must be called after it has been determined that there are no waiting orders, freeing the memory occupied by the result set.

The program then continues according to the pseudo code shown in Table 3. Generally speaking, the items in an order are retrieved next. For each item, its location is looked up and a route to retrieve that item is generated and then executed by the robot. After the last item is stored, the process is repeated for the next order. The database is updated continually to reflect the current state of the warehouse (stock levels, etc.) and orders (if an order is being picked or completed and stored in a drop–off bin). For example, the code below details how a stock update is performed:

```
var query_str : string in
    query_str := "update item set stock_level = ";
    call appint() (query_str, ?item_stock_level - ?item_quantity);
    call appstr() (query_str, " where item_id = ");
    call appint() (query_str, ?item_id);
    emit stock_query_out(query_str);
    await stock_results;
end var;
```

Table 3: Pseudo code describing the operation of the Esterel program running on the PC that accesses the database

```
loop forever
    if (orders are waiting) then
        Store the details of one order;
        if (free drop-off bin is available) then
            Store bin location;
            Set order status = PICKING;
            Get order lines;
            while (there are still order lines left to process)
                Robot picks up item(s) and drops in bin;
                Update the number of items in bin;
                Update item stock level;
            end;
            Set order status = STORED;
            Update bin with order number stored in it;
        end
    end;
    PAUSE
end
```

```
if (check_result(?stock_results)) then
    if (num_affected_rows(?stock_results) <> 1) then
        exit stock_update_failed;
    end if;
else
    exit bad_query;
end if;

call clear_results()(?stock_results);
```

The generation of the pick–up and drop–off routes is the only non–database–related function of the database access layer. A function `generate_route_to`, which is not displayed here but can be found in [26], generates a string consisting of *op codes* that represent the operations the robot must perform to pick up that item and to return to the communication point in the warehouse. The string is sent to the robot via a signal, and then another signal indicating the completion of executing the route is awaited. After the pick–up route is complete, a drop–off route is generated and executed in a similar fashion. By using external functions to generate the route, the warehouse can be re–designed in any way consistent with the item locations stored in the database, and only the two route generating functions will have to be re–written.

5.3.3 Route interpretation layer

To interpret a route string of op codes, a number of external C functions are provided that extract parts of the string; details can again be found in [26]. First, a function `get_num_ops()` is called to determine how many separate operations the route string contains. A loop is then repeated this number of times. On each iteration, a function `get_op()` is invoked to extract the type of operation and `get_param()` to extract the parameters to the operation. Once the operation type has been determined, an emission is made on the appropriate signal with the value obtained from `get_param()`. When the robot has completed the operation, either the `movement_op_complete` or `forklift_op_complete` signal is emitted, informing the route interpretation layer that the robot is ready for the next operation. When all operations have been performed in this manner, the signal `route_complete` is emitted, which lets the database access layer know that the robot has finished its route.

5.3.4 Hardware layer

The Movement RCX is required to run both the route interpretation layer and part of the hardware layer. To accomplish this, both layers are run in parallel and local signals are used to communicate between them. To communicate with the hardware layer running on the Forklift RCX, the output signals `pickup_item` and `drop_item` and the input signal `forklift_op_complete` are used (cf. Fig. 2). These signals, combined with the signals of the hardware layer running on the Movement RCX, i.e., signals `move_forward`, `turn_left`, `turn_right` and `movement_op_complete`, give the complete range of commands provided by the hardware layer.

6 Conclusions

This article presented two APIs for interfacing the synchronous programming language Esterel to the relational database MySQL. The *Local Result Set API* assumes the result set to a database query to be stored locally to a reactive Esterel kernel and largely relies on the external function concept of Esterel. The *Remote Result Set API* considers the result set to be stored remotely and is realised via Esterel's task concept. To the best of our knowledge, this article provides the first detailed discussion of, as well as APIs for, accessing relational databases from a popular, industry–strength language which is used for programming embedded controllers.

Both of our database APIs worked well in testing. In particular, the Local Result Set API is extensively used in our warehouse case study and, although none of the database operations there are particularly complex, they are representative of the kind of database operations performed by embedded systems. In contrast to the elegance exhibited by the Local Result Set API, the Remote Result Set API appears to be slightly more complex. This is due to the modelling of all database operations as tasks, which became necessary since remoteness implies that one cannot expect instantaneous responses and, hence, cannot employ external functions for accessing result sets.

The development of the Local Result Set API also showed that the language extension features provided by Esterel are indeed sufficient for our application. Given the constraints

imposed by a synchronous language, we feel that the developers of Esterel have selected an appropriate set of extensibility options that allow the language to be extended while keeping it within the synchronous paradigm. Of course, it is possible to abuse these features by allowing external functions and procedures to execute for a significant portion of time. However, if used correctly, the extensibility options allow great flexibility, from the ability to represent abstract data types to being able to pass these data types to external code.

It should be emphasised that the introduction of a database in an Esterel reactive system using either of our APIs does not undermine Esterel's synchrony hypothesis. However, since the response times for returning query results or for accessing remote result sets cannot be guaranteed, the system can end up waiting for a signal that may never arrive. If the database is one that can provide guaranteed response times, such as a real–time database, the problem is elevated. Otherwise, the issue must be solved via timeouts in system design. In our case study, all database operations are performed at non–time–critical points, whence any unexpected delay from the database simply results in the system pausing, not malfunctioning.

Finally, it must be mentioned that our approach is not restricted to the particular database MySQL employed by us. Indeed, our APIs can easily be adapted to those databases that support the *Open Database Connectivity* (ODBC) API [24], since the ODBC operations are quite similar to those provided by MySQL.

Acknowledgements. This research was supported by an Undergraduate Research Bursary of the Nuffield Foundation (grant no. URB/01528/G). We would also like to thank the anonymous reviewers for their detailed suggestions, as well as the participants of the SYNCHRON'04 workshop at Schloss Dagstuhl, Germany, for their constructive comments on an early version of this work.

References

[1] D. Baum, M. Gasperi, R. Hempel, and L. Villa. *Extreme Mindstorms – An advanced guide to Lego Mindstorms.* Apress, 2000.

[2] G. Berry. The constructive semantics of pure Esterel. CMA, École des Mines, INRIA, 1999. Draft version 3.0.

[3] G. Berry. The Esterel v5 language primer. CMA, École des Mines, INRIA, 2000.

[4] G. Berry. The foundations of Esterel. In *Essays in honour of Robin Milner.* MIT Press, 2000.

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] R.H. Bishop. *Modern control systems: Analysis and design using MATLAB and SIMULINK.* Addison Wesley, 1997.

[7] BrickOS. `http://brickos.sourceforge.net`.

[8] D.R. Butenhof. *Programming with POSIX threads.* Addison Wesley, 1997.

[9] P. Caspi, D. Pilaud, N. Halbwachs, and J.A. Plaice. LUSTRE : A declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*. ACM Press, 1987.

[10] Esterel compiler. `http://www-sop.inria.fr/meije/esterel/esterel-eng.html`.

[11] Esterel Technologies. Esterel Studio. `http://www.esterel-technologies.com`.

[12] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, 1993.

[13] D. Harel and M. Politi. *Modeling reactive systems with Statecharts: The STATEMATE approach*. McGraw Hill, 1998.

[14] I-Logix/Telelogic. Statemate. `http://www.ilogix.com`.

[15] National Instruments. NI LabVIEW Database Connectivity Toolkit. `http://www.ni.com/labview`.

[16] K.-Y. Lam and T.-W. Kuo. *Real-time database systems: Architectures and techniques*. Kluwer Academic Publishers, 2000.

[17] Lego Mindstorms robotics kit. `http://mindstorms.lego.com`.

[18] The MathWorks. Simulink. `http://www.mathworks.com`.

[19] The MySQL open source database. `http://www.mysql.com`.

[20] The MySQL C API. `http://dev.mysql.com/doc/mysql/en/C.html`.

[21] A. Pnueli and M. Shalev. What is in a step: On the semantics of Statecharts. In *Theoretical Aspects of Computer Science*, volume 526 of *Lecture Notes in Computer Science*, pages 244–264. Springer-Verlag, 1991.

[22] D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.

[23] T. R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference*, pages 328–333. IEEE Computer Society, 1996.

[24] R. Signore, J. Creamer, and M.O. Stegman. *The ODBC solution: Open database connectivity in distributed environments*. McGraw Hill, 1995.

[25] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, 2000.

[26] D. White and G. Lüttgen. Local and remote result set APIs. `http://www.cs.york.ac.uk/~luettgen/estereldb/`.

[27] D. White and G. Lüttgen. Accessing databases from Esterel. Technical Report YCS-2004-384, Department of Computer Science, University of York, England, 2004.