DCCL: Verification of Component Systems with Ensembles[*]

Jiří Barnat Nikola Beneš Ivana Černá Zuzana Petruchová
Faculty of Informatics, Masaryk University
Botanická 68a, 602 00 Brno, Czech Republic
{barnat,xbenes3,cerna,petruchova}@fi.muni.cz

ABSTRACT

Current trends in computing include building distributed systems out of autonomous adaptive components. Communication between the components may be local and communication channels may change over time. This emergent behaviour of communication may be seen as the creation and dissolution of component ensembles. Clearly, correctness of such systems is an important issue. We provide a verification-oriented modelling language for describing these component-ensemble systems as well as a verification tool. The processes of the components as well as the ensemble communication are described in a C++-like fashion. The tool is an extension of the parallel and distributed verification environment DiVinE. We also describe several demonstrative examples and use them to experimentally evaluate our approach.

Categories and Subject Descriptors

D.2.1 [**Requirements/Specification**]: Languages; D.2.4 [**Software/Program Verification**]: Model Checking

General Terms

Languages, Verification, Design

Keywords

component-based development, ensemble, adaptation, specification language, formal verification

1. INTRODUCTION

As software systems get larger, their development increasingly shifts from monolithic to component-based [6, 13]. The component-based systems are often built in a distributed fashion, where components are autonomous and even adaptive, if it is desirable for the system to work in changing environment. In such environments, communication channels may also change over time. This leads to the emergence of component ensembles, implicit groups of interacting components.

Let us consider, for example, autonomous cars moving in an environment with traffic rules. Each of these cars is supposed to repeatedly visit some designated places and it can communicate with other cars nearby. The design of the autonomous cars has to ensure that they follow the traffic rules, pass through crossroads without crashing into another car, etc. For efficiency it might also be desired that the cars form convoys whenever possible. The ensembles in this example might be the groups of nearby cars.

The concept of ensembles was introduced through projects InterLink [10] and ASCENS [1] to capture software-intensive systems with massive numbers of nodes, operating in open and non-deterministic environments, in which they have to interact and dynamically adapt to new requirements and conditions [7]. A mathematical model of ensembles and their composition has been introduced in [8] while an operational model of ensembles and a formal language SCEL that allows the description of ensembles in a compact and formal way was introduced in [4].

SCEL brings together various programming abstractions that permit to directly represent knowledge, behaviours and aggregations according to specific policies. The two central ingredients of SCEL are the notions of *autonomous component* and of *ensemble*. Ensembles are aggregations of components characterised by means of suitable predicates associated to the attributes *ensemble* and *membership*. Ensembles synthesise dynamically by exploiting the values of the components attributes and avoid the resorting to rigid syntactic constructs. Different dialects can be defined by appropriately instantiating the features of the language. One of the approaches that deal with the design of ensemble systems lead to the creation of the DEECo component model [11]. In their methodology, the authors propose separation of concerns by managing the ensembles as first-class entities, viewing the component interaction as data exchange that is not explicitly requested by the components themselves.

Our aim lies in the area of verification. Correctness of distributed component-based systems, while very important, may be hard to establish. The changing environment as well as the results of communication are non-deterministic in nature, which makes subtle errors such as race conditions both likely to happen and hard to find. Traditional methods of verification such as testing and simulation might not be enough and their coverage of the system's behaviour in-

[*]This work has been supported by the Czech Science Foundation, grant no. GAP202/11/0312.

sufficient. There thus rises a need for formal verification methods such as model checking. To be able to perform these methods, we need a concrete modelling language with precisely specified syntax and semantics. At the same time, the desired modelling language should be close to real-world programming concepts.

In this paper, we present the Dynamically Communicating Components Language (DCCL), a verification-oriented modelling language that implements the concepts found in the SCEL language [4] and in the DEECo component model of [11]. This formalism gives precise formal semantics to the concepts of components and ensembles. The system consists of components, which can perform some computation independently. Each of these components has some local knowledge representing its state. This knowledge can be read by the component and used in its computations.

The components do not communicate directly by sending and receiving messages. Instead, they form ensembles. Every ensemble contains a membership predicate that decides whether a particular component at a particular point in time belongs to that ensemble or not. Once a set of components form an ensemble, this ensemble ensures implicit communication between them.

The computation of the system is divided into two alternating phases: the component phase, where components are performing their computation, and the ensemble phase, where ensemble predicates are evaluated and implicit communication is performed. In the component phase, all component computations are performed independently.

As for the ensemble phase, we present three different kinds of semantics. Each of the semantics represents different view of the relative speed of communication and component computation. The first one is the *fixpoint* semantics, in which ensemble communication steps are performed until a fixpoint of all components' knowledge is achieved. The second one is the *time-unit* semantics, in which the number of ensemble steps that are performed between successive component phases is limited by a predefined number. The third one is the time-unit semantics with *broadcasting*, which differs from the previous one by allowing the existence of broadcast ensembles, whose communication is treated slightly differently with respect to the time-unit limit.

For the specification of the properties we want to verify on a DCCL model, we use the Linear Temporal Logic (LTL, [12]). This logic allows to write statements about all possible runs of a system, using temporal operators such as *always*, *eventually*, *until*, and their combinations. The DiVinE environment [2, 5] we base our verification tool on then uses the standard automata-based approach to LTL verification [14].

The rest of the paper is organised as follows. In Sections 2 and 3 we describe the syntax and semantics of DCCL, respectively. Section 4 then describes the verification process in detail. In Section 5 we present some demonstrative examples and show experimental results. The paper's conclusion and an overview of future work is then given in Section 6.

2. DCCL SYNTAX

We present the syntax of DCCL on two levels: abstract syntax, which is the mathematical formalism behind DCCL, and concrete syntax, which is the C++-like language used to write the models. The main difference is that the abstract syntax operates with notions such as functions without explicitly defining how a function is to be described, while the concrete syntax works with actual C++-like code. Separating these two levels has the advantage of abstract syntax reusability. If we wanted to implement DCCL on top of a language other than C++, we would only need to modify the concrete syntax.

On the abstract level, a DCCL model consists of three parts: the semantics specification, the components' description, and the ensembles' description. The semantics specification is either *fixpoint* or *time-unit*(t) where t is a natural number parameter. The meaning of this specification is going to be explained in the next section. In the concrete DCCL syntax, the model consists of the three parts mentioned above plus a part consisting of auxiliary definitions that introduce user-defined types and functions as well as some primitives used for verification. The semantics specification is given at the beginning of the model file using either `semantics fixpoint;` or `semantics timeunit` t`;`, where t is a positive number literal. Alternatively, the semantics specification may be omitted from the input file—the semantics is then specified when invoking the tool, see Section 4.

2.1 Component

The central notion is that of *component type*. A component type consists of its *knowledge type* and its *process*. On the abstract level, the knowledge type is a finite set of all possible states of the component, or its knowledge. The knowledge can be changed either by running the process of the component, or by the ensemble communication. We denote by K_C the knowledge type corresponding to component type C.

In the concrete DCCL syntax, the knowledge type is represented by a set of fixed-size C++ variables, of built-in or user-defined `struct` types. The definition of each of the component types comes first, the actual components of that type are later instantiated by stating the initial knowledge of each component, see Fig. 1.

```
component car {
    int posx, posy;
    int direction;
    int move;
}
```

```
car {
    posx = 3; posy = 2;
    direction = 2;
    move = 1;
}
```

Figure 1: A component type and its instantiation.

The *process* of a component is a periodically repeating computation that modifies the component's knowledge. The process corresponding to component type C, denoted by P_C may be thus seen, on the abstract level, as a function $P_C : K_C \to K_C$.

In the concrete DCCL syntax, the process is defined as a C++-like function that can access and modify all elements of its component's knowledge. The function is supposed to be side effect free. For an illustration, see Fig. 2.

2.2 Ensemble

An ensemble description consists of a *membership predicate* p and a *mapping function* m. An ensemble instance then is a set of components with one coordinator c and arbitrarily many members such that $p(c, d)$ holds for every

```
process car {
    if (move)
        switch (direction) {
            case 1: posx++; break;
            case 2: posy++; break;
            case 3: posx--; break;
            case 4: posy--; break;
        }
}
```

Figure 2: The process of a component.

member d. Whenever an ensemble is formed, knowledge between its coordinator and its members is exchanged using the mapping function. This function operates on each pair (coordinator, member) separately. We may thus write (using $\mathcal{C}$ to denote the set of all components and $\mathcal{K}$ to denote the set of all possible knowledge):

$$E := \langle p, m \rangle$$
$$p : \mathcal{C} \times \mathcal{C} \rightarrow \{true, false\}$$
$$m : \mathcal{K} \times \mathcal{K} \rightarrow \mathcal{K} \times \mathcal{K}$$

In the concrete DCCL syntax, the ensemble is defined by stating a set of component types which can be the coordinator, a set of component types which can be the member and writing the predicate and mapping. The membership predicate is written as the contents of a C++ if-condition and the mapping function is the code executed if this condition holds. The code may assume the existence of two pointers `c` and `m` that point to the knowledge of the coordinator and the member, respectively. It is possible to use user-defined functions (see the following subsection) in both the predicate and the mapping.

It is also possible to write several disjoint if-conditions since this is equivalent to encapsulating the whole code in one if-condition that is a disjunction of the original conditions. For an example, see Fig. 3.

```
ensemble Cross {
    crossing;
    car;
    if (is_in_path(c->posx, c->posy,
            m->posx, m->posy, m->direction)
        && c->blocked && m->move) {
        m->move = 0;
    }
    if (c->posx == m->posx
        && c->posy == m->posy && !m->move) {
        c->blocked = 1;
    }
}
```

Figure 3: An ensemble description.

Apart from the general ensemble description, we also have *broadcast* ensembles, whose function is explained in the next section. The broadcast ensemble is also defined by a *membership predicate* p and a *mapping function* m. The role of p and m is similar to previous, only here, the mapping function may only modify the knowledge of the ensemble's members and not its coordinator's. We may thus write:

$$E_{\mathrm{B}} := \langle p, m_{\mathrm{B}} \rangle$$
$$p : \mathcal{C} \times \mathcal{C} \rightarrow \{true, false\}$$
$$m_{\mathrm{B}} : \mathcal{K} \times \mathcal{K} \rightarrow \mathcal{K}$$

In the concrete syntax, the definition of a broadcast ensemble is similar to the general ensemble member ensemble. We specify an ensemble it to be of the broadcast kind by including the word broadcast as a substring of the ensemble name. Obviously, as a broadcast ensemble may not modify the knowledge of its coordinator, the data pointed at by the pointer `c` is read-only.

2.3 Auxiliary DCCL Constructs

Apart from the description of component types, ensembles, and the component instances, the concrete DCCL syntax also allows several auxiliary constructs. These syntactic elements have no direct impact on the dynamic semantics of the model, but they simplify the modelling task and/or provide helpful functions that are used in the phase of model validation and verification.

First of all, it is possible to define custom `struct` types that work the same way as in C++. As already mentioned, these can then be used as a part of the definition of knowledge type. The user-defined types are also used to define static global data. This is done by specifying one of the `struct`s as the type of this global data and by writing an initialisation function. Once the data is initialised at the beginning of the system's run, it may be accessed by both components and ensembles (they may assume the existence of a pointer to the global data called `global`), but may never be modified.

```
struct world {
    int map[2][7];
}

global world;

init {
    int tmp[2][7] = { {0,1,1,0,1,0,1},
                      {1,0,1,1,0,1,0} };
    for (int i = 0; i < 7; i++){
        global->map[0][i] = tmp[0][i];
        global->map[1][i] = tmp[1][i];
    }
}
```

Figure 4: Declaration and initialisation of static global data.

Auxiliary functions that can be then used in the component's processes, in the ensemble descriptions and in the initialisation of the global data can be defined in a C++-like way, prepending the function header with the keyword `global`.

In order to perform the LTL verification task, it is needed to specify the atomic propositions. In DCCL this is done by writing a function which uses both the state and the global variable, the global phase and round, and returns true or false. An example of such function is given in Fig. 5. The function is defined using the keyword `prop` and the name of the atomic proposition. The function may assume the existence of a pointer `in` to the actual state of the system. The state of the system consists of the following: a positive integer `round`, a Boolean value `phase`, and an array called `things` for every component type `thing` that contains the knowledge of every component (ordered by their appearance in the DCCL source code) plus a positive integer `timeunit`. The role of `round`, `phase`, and `timeunit` is explained in the next section.

```
prop crash {
  for (int i=0;i<3;i++)
    for (int j=i+1;j<3;j++)
      if (in->cars[i].posx == in->cars[j].posx
       && in->cars[i].posy == in->cars[j].posy)
        return true;
  return false;
}
```

Figure 5: Atomic property specification.

For the purpose of using DiVinE to draw the trace of the generated state graph or a counterexample, it is possible to specify which parts of component's knowledge are to be displayed by writing the `print` function, which consists of either a single `printf` call or an arbitrary C++ code that uses a `stringstream` object called `output`. The code can refer to the component knowledge, its `timeunit` or the `global` variable. See Fig. 6.

```
print car {
    printf("x: %d y: %d dir: %d time: %d",
            posx, posy, direction, timeunit);
}
```

Figure 6: Component knowledge output format.

3. DCCL SEMANTICS

The computation of the system works in two alternating phases, the *component phase* and the *ensemble phase*. In the component phase, components perform their computation as prescribed by their process description. After the component phase, the system switches to ensemble phase, where ensemble communication is performed. Every ensemble communication step is atomic. The way of forming ensembles and performing ensemble communication steps varies slightly in the three types of semantics presented.

Formally, we define a labelled transition system $(\Sigma, \mathcal{L}, \to)$, where Σ is a set of states, $\mathcal{L}$ is a set of labels and $\to \subseteq \Sigma \times \mathcal{L} \times \Sigma$ is a labelled transition relation.

The definition of states depends on the semantics we want to use. In the fixpoint semantics, a state simply consists of the knowledge of every component. That is, if we have components $c_1, c_2, \ldots, c_n$ where the component type of c_i is C_i, then the set of states is:

$$\Sigma_\mathrm{f} = K_{C_1} \times \cdots \times K_{C_n} \times \{\mathbf{C}, \mathbf{E}\}$$

That is, the state of the system consists of the knowledge of all components and a specification of the current phase ($\mathbf{C}$ for component phase, $\mathbf{E}$ for ensemble phase).

In the time-unit semantics, we add to each component a time counter whose maximal value t is given by the semantics specification. The set of states can be then written as follows:

$$\Sigma_t = K_{C_1} \times \cdots \times K_{C_n} \times \{0, \ldots, t\}^n \times \{\mathbf{C}, \mathbf{E}\}$$

In the description of the semantics, we use the following notation. Whenever we have a state $\sigma \in \Sigma_\mathrm{f}$, we use σ_i to denote the knowledge of the ith component and $\sigma[i \mapsto k']$ to denote the state that is created from σ by changing the knowledge of the ith component to k'. For a state $\sigma \in \Sigma_t$, we have similar notation: σ_i is the pair (k_i, t_i) of knowledge and remaining time units of the ith component; further, $\sigma[i \mapsto (k', t')]$ is the state that is created from σ by changing the knowledge of the ith component to k' and its remaining time units to t'. We also use $\sigma[\mathbf{C}]$ and $\sigma[\mathbf{E}]$ to denote the state that is created from σ by changing the phase indicator to $\mathbf{C}$ or $\mathbf{E}$.

3.1 Component Phase

In the component phase all components perform their processes. These processes are independent—communication only happens in the ensemble phase and the global data are read-only. We can thus safely assume that all the components' processes happen at the same time, all at once. Let now $k'_i = P_i(\sigma_i)$ be the effect (the modified knowledge) of the ith component process. In the fixpoint semantics, let $\sigma' = \sigma[1 \mapsto k'_1, \ldots, n \mapsto k'_n]$; in the time-unit semantics, let $\sigma' = \sigma[1 \mapsto (k'_1, t), \ldots, n \mapsto (k'_n, t)]$, where t is the maximal time limit. The only transition of the component phase is then: $\sigma \xrightarrow{comp} \sigma'[\mathbf{E}]$.

3.2 Ensemble Phase

The basic unit of the ensemble phase is that of ensemble step in which two components satisfying the component predicate of an ensemble get their knowledge modified by the mapping function of the ensemble.

3.2.1 Fixpoint Semantics

In the first semantics, the ensemble phase runs until a fixpoint is reached, where no more possible ensemble steps can be performed. In that state, every component has all the knowledge it can gain about the system through ensemble communication, and another component phase starts.

Let $E = \langle p, m \rangle$ be an arbitrary ensemble description. Let further c_i and c_j be two components such that the membership predicate $p(c_i, c_j)$ holds. This means that c_i is a coordinator of an ensemble and that c_j is one of its members. Let now k_i, k_j be the knowledge of c_i, c_j, respectively, and let $m(k_i, k_j) = (k'_i, k'_j)$ be the result of the application of the mapping function. The transition corresponding to this mapping may be written as follows:

$$\frac{\sigma' = \sigma[i \mapsto k'_i, j \mapsto k'_j] \qquad p(c_i, c_j) \qquad m(\sigma_i, \sigma_j) = (k'_i, k'_j)}{\sigma \xrightarrow{ens(E, c_i, c_j)} \sigma'}$$

If no more ensemble communication steps can be performed (either because there are no components satisfying the predicate of any ensemble, or the application of the mapping function results in no change in the knowledge), the ensemble phase ends.

$$\frac{\nexists E, c_i, c_j : \sigma \xrightarrow{ens(E,c_i,c_j)} \sigma'}{\sigma \xrightarrow{end} \sigma[\mathbf{C}]}$$

3.2.2 Time-Unit Semantics

In this semantics, the number of steps of the ensemble phase is limited. We do this by dividing the ensemble phase into several rounds. In each round, multiple ensemble mappings may occur but with the limitation that each component can participate in at most one of such mappings. We enforce this behaviour using time units that are assigned to each component. At the beginning of the ensemble phase, each component has t time units, as specified by the semantics specification.

The rounds are numbered t, $t-1$, $t-2$, ..., 1. In each round ℓ, ensemble communication may occur, similarly to previous, if both participating components still have ℓ time units. The ensemble mapping transition for the ensemble description E, components c_i, c_j and round ℓ is given as follows:

$$\frac{\begin{array}{ccc}\sigma_i = (k_i, \ell) & \sigma_j = (k_j, \ell) & m(k_i, k_j) = (k'_i, k'_j) \\ p(c_i, c_j) & \multicolumn{2}{c}{\sigma' = \sigma[i \mapsto (k'_i, \ell-1), j \mapsto (k'_j, \ell-1)]}\end{array}}{\sigma \xrightarrow{ens_\ell(E,c_i,c_j)} \sigma'}$$

Once there are no ensemble communication steps to be performed in round ℓ, the round ends. If $\ell > 1$, the next round begins and every component's time unit counter is reduced to $\ell - 1$.

$$\frac{\nexists E, c_i, c_j : \sigma \xrightarrow{ens_\ell(E,c_i,c_j)} \sigma'}{\sigma \xrightarrow{round} \sigma[\forall x : x \mapsto (\sigma_x, \ell-1)]}$$

If $\ell = 1$, the ensemble phase ends. Every component's time unit counter is reset to t.

$$\frac{\nexists E, c_i, c_j : \sigma \xrightarrow{ens_1(E,c_i,c_j)} \sigma'}{\sigma \xrightarrow{end} \sigma[\mathbf{C}, \forall x : x \mapsto (\sigma_x, t)]}$$

3.2.3 Time-Unit Semantics with Broadcasting

This semantics extends the previously mentioned one with the possibility to use broadcast ensembles and treat them in a distinct way. Intuitively, the difference is that whereas in the time-unit semantics, the coordinator of an ensemble has spent one time unit per each knowledge exchange with its member, the coordinator of a broadcast ensemble only spends one time unit per each broadcasting step. The broadcasting step modifies the knowledge of all members of the ensemble that still posses an unused time unit.

Formally, let $E = \langle p, m \rangle$ be a broadcast ensemble, let c_i be the coordinator component and let $c_{j_1}, \dots, c_{j_s}$ be the member components, i.e. $p(c_i, c_j)$ for every $j \in \{j_1, \dots, j_s\}$ and it is the maximal set satisfying this condition. Let further $m(k_i, k_j) = k'_j$ where k_j is the knowledge of c_j. The broadcasting transition can be then written as follows ($\forall j$ is qualified over $\{j_1, \dots, j_s\}$):

$$\frac{\begin{array}{ccc}\sigma_i = (k_i, \ell) & \forall j : \sigma_j = (k_j, \ell) & \forall j : m(k_i, k_j) = k'_j \\ \forall j : p(c_i, c_j) & \multicolumn{2}{c}{\sigma' = \sigma[i \mapsto (k_i, \ell-1), \forall j : j \mapsto (k'_j, \ell-1)]}\end{array}}{\sigma \xrightarrow{bcast(E,c_i)} \sigma'}$$

4. VERIFICATION PROCEDURE

For the new language described in the previous section we also deliver simulation and verification tools to help software engineers create faithful and reliable models of complex real software solutions. Our tool builds on top of the parallel and distributed-memory explicit-state model checking tool DiVinE [2, 5].

Model checking [3] is a formal verification procedure that takes two inputs – model of a distributed (component-based) system and one particular property the system should meet. For these inputs, the model checking procedure decides automatically whether the given model satisfies the given property or not. For the purpose of model checking the properties to be checked for must be formalised using temporal logic formulae. In our case, we use Linear Temporal Logic (LTL) formulae and employ automata-based approach to LTL model checking [14].

Our newly implemented tooling solution employs the so called Common Explicit-State Model Interface (CESMI) of the DiVinE model checker that allows for tool chaining of the model checker with third-party language interpreters. CESMI prescribes a simple binary interface between the verification core of the model checker and a model to be verified. By binary interface we mean that to access a model via CESMI a set of accessor functions must be implemented and provided to DiVinE in a form of dynamic (shared) library, the so called CESMI module. For a CESMI compliant module DiVinE offers various services, primarily formal verification by means of automata-based LTL model checking. Other services include discrete step-by-step simulation of the system under investigation and/or visualisation of the underlying reachable state space graph.

The workflow to use DiVinE on top of DCCL models is as follows. First, a DCCL model is taken from a text file and processed with our `dccl2cesmi` tool (available at [9]) into a C++ source file of the corresponding CESMI module.

```
$ dccl2cesmi model.dccl
```

Optionally, the semantics specification may be also provided (either `fixpoint` or a positive integer). This specification overrides the one given in the input file.

```
$ dccl2cesmi model.dccl 10
```

The resulting C++ file is then compiled with DiVinE into a dynamically loaded library using DiVinE's compile command. Any LTL properties to be verified for the model must be provided in a separate text file and passed to the compile command as an additional parameter at this time.

```
$ divine compile -cesmi model.cpp model.ltl
```

The compilation creates a new `model.so` file that can be used by DiVinE as the primary input file. Useful DiVinE commands to work with such a model are the following:

- `info` – List properties of a given model that can be checked by DiVinE. These properties include individual LTL formulas as given to the compile command in the previous step.
- `verify` – Perform verification of a given property. Note that the type of verification task is automatically chosen accordingly with respect to the property type. If

```
$ divine verify model.so
property deadlock:  deadlock freedom
---------------- Reachability ---------------
  searching...
  125302 states, 151042 edges, DEADLOCK
============================================
         The property DOES NOT hold
============================================

$ divine verify -r model.so 2>&1|grep CE-Init
CE-Init:  1,1,2,2,1,2

$ divine simulate -trace=1,1,2,2,1,2 model.so
```

Figure 7: Example of DiVinE invocation.

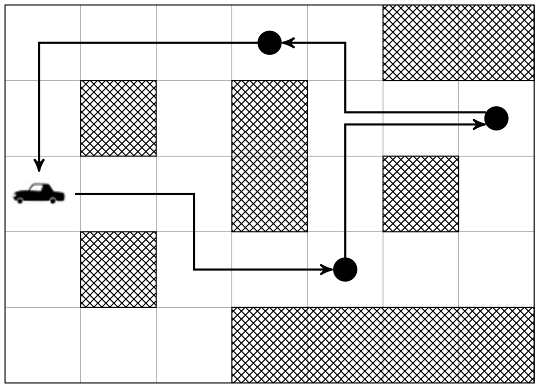

Figure 8: Autonomous cars example illustration.

an LTL formula is specified as a property, DiVinE performs LTL model checking; if a test for deadlock freedom is selected, DiVinE performs search for deadlock states.

- `simulate` – Runs a text-based simulator that allows to either interactively guide the simulator through the state space of a given model, or to execute and print a predefined run (sequence of actions) in the state space graph of the model.
- `draw` – Visualise (part of) the state space of a given model.

An example of particular invocation of DiVinE verify command is given in Figure 7. First, `verify` command is executed for a system given as CESMI module `model.so`. DiVinE informs user that it is about to verify deadlock freedom property for the system, and starts the verification process (search for a deadlock). When it detects a deadlock, it reports the number of states and transitions that have been explored, and informs user about the invalidity of the verified property.

Since the property is violated, user might get interested in the reasons why it is so. To that end, the model checker is capable of generating the so called counterexample – a run of the system that witnesses the property violation. Therefore, the example in Figure 7 proceeds with a new execution of DiVinE, this time with parameter $-r$ instructing DiVinE to produce a detailed report on the verification process. Among others, the report includes the counterexample described as a sequence of numbers. These numbers refer internally to actual transitions in the system that must be fired in order to get the system from an initial state to a deadlock state. For inspecting the counterexample in a human readable form, simulate command of DiVinE may be used. Figure 7 shows a particular invocation of DiVinE to do so.

5. EXAMPLES & EXPERIMENTS

To demonstrate the modelling and verification capabilities of our language and our verification tool we present two toy examples. The first, more elaborated, example concerns autonomous cars moving around in an environment with traffic rules, as mentioned in the introductory section. The second, simpler example then shows how the verification task can also be used to solve a different kind of problem, namely that of finding a strategy for control synthesis. After describing the examples and discussing the resulting models and their variants, we provide some experimental evaluation of the tool, showing the verification results.

5.1 Autonomous Cars Example

Let us now imagine a closed traffic environment with roads and crossroads. There are several autonomous cars that move around in this environment. Each has a predefined set of waypoints it is supposed to visit in a repeated fashion. While doing so, the cars are also supposed to respect the traffic rules, avoid crashing into another cars, and form convoys whenever possible. To achieve this, the cars are equipped with a short-range communication device that can be used to communicate with other nearby cars.

In order to model this system in DCCL, we need to decide on the kinds of components and ensembles we are going to use. Instead of having just the autonomous *car components*, we also introduce *crossroads components* that are going to help coordinate the car movement. These components are going to keeps track of cars that are waiting on each of the crossroads' incoming roads. They are then going to allow cars to cross in a FIFO manner. The crossroads only takes the first car in each direction into account, so if there are cars waiting at the crossroads on each of the roads, it lets them cross in a round-robin fashion, ignoring the number of waiting cars.

As for the ensembles, we have two ensemble types—the *crossroads ensemble* that facilitates the communication between a crossroads component and the components corresponding to adjacent cars; and the *convoy ensemble* that is going to form whenever there are several cars in a straight line wishing to move in the same direction.

Our modelling and verification approach only deals with finite-state systems. We thus view the traffic environment in a discrete manner as a square grid plan. The movement of the cars is then modelled as a discrete position change that is performed in each of the car components' processes.

In more detail, the global data and the components' knowledge is as follows:

- *Global data* includes the map of the traffic environment and the travel plan of each autonomous car. Although keeping the travel plan of each car in that car's knowledge would be more realistic, we do this to decrease the size of the model's states, which only depends on the size of the components' knowledge. This decision does not lead to any incorrect behaviour, as we assume that the cars' travels plan do not change throughout the run of the system.

The map and the plans are precomputed in the `init` function from a set of crossroads descriptions and a set of waypoints for each car. This information is currently hard-wired into the model; however, the model could be easily extended with the possibility to read such data from an external file.

- *The `car` component's* knowledge includes the following information: The position of the car (`posx`, `posy`), its direction (`dir`) and a flag whether the car is currently moving (`move`). The car's process moves the car, i.e. changes its position, if `move` is set to true, otherwise it remains in the same place. Then, regardless of whether it moved or not, it resets `move` to true again to try to go forward in the next component phase.

- *The `crossroads` component's* knowledge consists of its position (`posx`, `posy`), a flag indicating whether the crossroads is currently blocked by a car (`blocked`), the next direction from which the crossroads component is going to allow a car to cross (`freeDirection`), and two arrays representing whether cars are waiting in each incoming direction and the time they have been waiting (`waiting[4], timeWaiting[4]`). The crossroads' process keeps track of the waiting time and clears the blocked flag—if a car is still blocking the crossroads, the flag will be set again in the next ensemble phase.

```
process crossroads {
    for (int i=0;i<4;i++)
        if (waiting[i]) timeWaiting[i]++;
    blocked = false; freeDirection = 0;
}
```

We may now turn our focus towards the two kinds of ensembles.

5.1.1 Crossroads Ensemble

The `Cross` ensemble serves to ensure that no two cars enter given crossroads at the same time. The coordinator of this ensemble is always a crossroads component, while its members are the components corresponding to the cars waiting at or currently moving through the crossroads. This implies that at any point of time, there can be at most as many crossroads ensembles as there are actual crossroads in the environment.

The mapping function of this ensemble works as follows:

- If a member car is currently standing directly on the crossroads and has its `move` flag cleared, i.e. it is not going to move in the next component step, the crossroads becomes blocked.

- If one or more cars are waiting on the crossroads and the crossroads is not blocked, the ensemble lets one of these cars pass, marking the direction as free. The chosen direction is the one with the longest waiting time. If more directions have the same waiting time one of them is chosen nondeterministically. After marking one direction as free, the ensemble resets its waiting time.

  Since the nature of ensemble communication is nondeterministic, it can happen that a crossroads realises it is blocked after it decided to let one car pass. Therefore, to avoid resetting the waiting time for the longest waiting direction (and thus starvation), the crossroads saves the longest time for this purpose.

- If a car comes to the crossroads or is standing near, and either the crossroads is blocked, or the car's incoming direction has not been chosen as described above, the car is stopped, i.e. its `move` flag is cleared.

- If a car comes to the crossroads and no other car is waiting there, blocking the crossroads or has been already chosen to be able to cross, let this car pass. (This is a special case of the second bullet point.)

- If the crossroads became blocked and a car was already chosen to pass, set back the waiting time to the value mentioned above and stop the car.

The ensemble works by only stopping the cars, but after a car is marked to not move, it is never set back to move with this ensemble, therefore the computation stops after a finite number of moves.

5.1.2 Convoy Ensemble

Adding the possibility to form convoys serves to improve the movement efficiency of the autonomous cars in the following situation. Consider a car that has another car directly in front of it. As we want to avoid collisions, the only safe behaviour in such situation is to stop and wait until the position in front of the car becomes empty. Such solutions may, in a situation with many cars moving in a straight line, result in unnecessary stopping of cars.

We thus use the convoy to allow the cars to share the information about their moving intents thus allowing a car to move to an occupied position if it is sure that the occupied position will simultaneously become available during the next component step. The convoy ensemble also has to ensure that once one of the cars in the convoy decides to stop, all subsequent cars are also going to stop.

Instead of just one ensemble that would encompass all cars moving in a straight line, we employ several smaller ensembles, each containing only two cars moving behind one another. All this ensembles are going to be of one ensemble type, `Behind`. This ensemble forces a car to stop whenever there is a car in front of it that does not want to move in the next component step, i.e. it has its `move` flag cleared. Clearly, in the case of a convoy bigger than just two cars, the information about the first car stopping has to use some time to propagate using the ensemble communication from the first car to the last. This serves us as a demonstration of the effect of the time-unit limit on the correctness of the system, as will be seen in the Experiments section.

5.1.3 Adding Nondeterministic Components

Although the component processes in DCCL are deterministic, the knowledge exchange during the ensemble phase is inherently nondeterministic. We may thus exploit this fact to demonstrate the possibility of adding components with nondeterministic behaviour. Such components may be useful for the modelling of unpredictable environmental constraints.

In our autonomous cars model, the role of an unpredictable entity is going to be played by a pedestrian component. This component is going to represent a nondeterministically moving obstacle running over which is forbidden. The knowledge of `pedestrian` contains its position and direction. Its process moves the pedestrian by one step in the chosen direction. The choice of the direction, however, happens in the ensemble phase. In order to do that, we define a helper component and four ensembles, one for each of the directions. These ensembles use the helper as the coordinator and the pedestrian as the member. The helper can be used only once each ensemble phase. One of the ensembles is nondeterministically chosen and the pedestrian's direction is set to this direction.

We further include a broadcast ensemble, of which the pedestrian is a coordinator. The ensemble's members are the cars standing on the same part of the road as the pedestrian and the broadcasting function causes them to stop.

5.1.4 *Verification*

To demonstrate the verification capabilities of our tool, we have chosen two important properties of the autonomous cars model. The first one is a safety property that ensures that the cars do not crash or run over the pedestrian. The property is written using LTL in DiVinE's property notation as `G !crash`, where `G` represents the *always* operator, `!` represents negation, and `crash` is the atomic proposition that holds whenever two cars share a position or a car shares its position with the pedestrian.

The second property is a liveness property: we would like to ensure that a car reaches some of its waypoints then it eventually reaches its next waypoint. This could be done straightforward by stating a a conjunction of several formulae of the form `G(car1wp1 -> F car1wp1)`, where `F` is the *eventually* operator, `->` represents implication, and `car1wp1` is the atomic proposition that holds whenever car no. 1 stands on its first waypoint, with similar propositions defined for other cars and waypoints. However, the movement of the cars in our system is cyclic, therefore the property `GF(car1wp1)`, checking whether the car no. 1 can always reach its first waypoint in the future, is equivalent to the above conjunction of formulae, therefore we chose to use the simpler version.

5.2 Burglar Example

With this simple model we demonstrate the possibility to use model-checking and the counterexample for strategy synthesis.

The model is a square map with a treasure in the middle. There are guards, which circle around this treasure on square paths. There is also one burglar, who can nondeterministically move in four directions or not move at all if she chooses to. The goal of the burglar is to reach the centre of the map without being seen by any of the guards.

We model the burglar as a component whose knowledge contains the burglar's position and the desired direction of her next movement. As in the case of the pedestrian component in the previous example, the process of this component only ensures the movement while the changing of direction happens nondeterministically in the ensemble phase, using helper component and five kinds of ensembles, one for each direction and one for the choice of no movement.

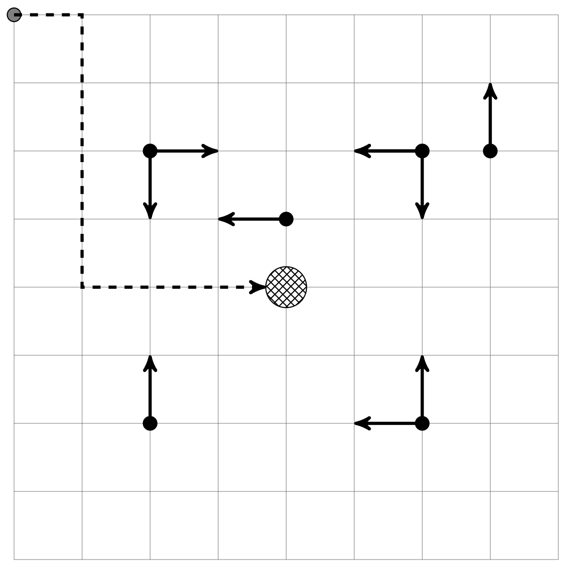
Figure 9: Burglar example illustration.

The guard is a component whose knowledge contains its position, direction and distance from the treasure. Its process moves the guard in the direction it is headed and then rotates the guard, if it is in the corner of its path. The guard does not take part in any ensemble.

We define three atomic propositions. The first atomic proposition `seen`, true if the burglar is seen by one of the guards. The burglar is seen, if the guard is looking at her, i.e. when the burglar is at the same line as the guard and the guard is heading towards the burglar. The other proposition, `centre` is true if the burglar is in the centre of the map (at the treasure's location).

The formula we want to verify is `F(seen) || G(!centre)`, representing the property that no run of the system should be such that the burglar is never seen and yet is able to move to the treasure's location. If this formula holds, the burglar may not steal the treasure unnoticed. However, if the formula does not hold, we are even given a counterexample, i.e. detailed instructions for the burglar about the way she should move in order to achieve her goals.

5.3 Experimental Results

We summarise the performance of DCCL on the three models described above. We focus on the first two models first with the results for the Burglar model stated separately.

- *Carplan* is the square grid map with cars moving on it. The cars follow a plan of visiting waypoints computed at the start of the run and form ensembles in convoys and on crossings along their paths if they meet.

- *Pedestrian* is the Carplan model with an added pedestrian. This pedestrian moves each round in a random direction and stops cars at her actual position from moving by forming a broadcast ensemble with them.

- *Burglar* is the model of the burglar trying to pass the guards to reach the treasure at the centre of the map.

We measured the state space of each model—see the rows of Table 1 with no property formula. As explained in Section 3, the only branching in DCCL occurs in the ensemble phase. In the Carplan model, most of the time there is some branching in one ensemble phase, which ends in only a few different outcomes by the end of the ensemble phase. This is especially true, if there are enough time units to ensure the cars not crashing.

By adding the pedestrian into Carplan to get the Pedestrian model, the size of the state space get increased a lot, since the pedestrian blocking a car causes all mutual (non-crashing) positions of the cars along their paths to be possible, in addition to enlarging the state space by all the possible pedestrian positions.

For these two models, we checked the following two properties:

- `G(!crash)`—a safety property stating that the cars do not crash according to the rules mentioned in the previous section; and
- `GF(car1wp1)`—a liveness property stating that car 1 visits its first waypoint infinitely often. Since the cars move in a cycle, this checks whether the car always moves forward eventually.

We used the cutoff values of time or car numbers for the tests. We also used the fixpoint model for the verification to show that with enough time units and no broadcast the same properties hold both for the fixpoint and timeunit semantics.

As for the `G(!crash)` property, at least 7 time units are needed with the 4-car variant of the model for it to hold. With 6 time units, the cars will crash, if they are positions [3,2], [2,2], [1,2] and [2,1] heading north, south, south and east, respectively. In one of the possible runs, the cars get blocked by each other in the order car3, car1, car2 and car4. However, since some operations of the crossing ensemble take more than one turn, in 6 rounds the information to stop doesn't propagate to the last car, which in the next component phase crashes into the second car.

As for the `GF(car1wp1)` property, if there are 4 cars moving in the Carplan model, they can get stuck on positions [2,1], [2,2], [3,2] and [3,3] heading east, south, north and west, respectively. If the crossing at [3,2] first chooses car 4 to pass, car 2 becomes blocked by this, since it cannot move to the same crossing. This, however, blocks the crossing at [2,2] and in turn blocks the third car. With both crossings blocked with unmovable cars, no car can move anymore and a deadlock occurs.

With 3 cars present, this could also be an issue, since the first car doesn't have to take part in the deadlock situation. However, with less cars, there are less reachable combinations of car positions, therefore this situation never occurs.

In the People model, two cars suffice for the liveness property to not hold. Again at crossings [2,2] and [3,2], if the first car stands on the north crossing and wants to go south, while the second car stands on the south crossing and wants to go north, the pedestrian standing on either of these two places blocks both cars from moving in the next component phase. Since the pedestrian can switch between these two positions, she can block the two cars indefinitely.

The results are summarised in Table 1.

For the Burglar model, we ran the experiments on maps of two sizes, with two guard placements for the smaller map size. On the smaller map with 9 guards, they are initially placed as illustrated in Figure 9, with arrows pointing from marked spots representing the guards and their initial direction. For the second model we added one guard, so that it is not possible for the burglar to reach the centre unseen anymore. The model on the bigger map serves to display the state space increase.

On the 8x8 map with 9 guards, by verifying the formula `F(seen) || G(!centre)` we obtain the result that the property doesn't hold, along with the counterexample run which represents the strategy for the burglar. Clearly, the one guard missing in the middle circle means there is almost always the possibility to cross the guards in this circle and the other two guards cannot compensate this.

By adding the last guard to the middle circle, there is always a guard looking on [2,2] and west from this point, and the same holds for the north direction. This means, that the burglar starting at [0,0] cannot move anywhere than on the four places nearest to him without being seen. The verification results indeed correspond.

On the bigger map, the guards are in such positions that the burglar is again able to reach the centre.

To verify the formula or to obtain a counterexample we do not need to search the whole state space. This can be seen on the 8x8 map with 9 guards, where number the states searched on the product automaton is almost the same as the number of states of the system, meaning only a portion of the system states had to be used. This is even more visible on the 10x10 map, where it is easier for the burglar to reach the centre due to the initial position of the guards, and in the 8x8 model with 10 guards, where the burglar can move only within four positions.

This model does not use broadcast, therefore it is possible to run with fixpoint semantics instead of timeunit, further reducing the state space. The results of the verification are the same in fixpoint semantics and in the timeunit. They are summarised in Table 2.

All of the models used in the experiments presented in this section are available at the DCCL web page [9].

6. CONCLUSION & FUTURE WORK

In this paper, we have presented DCCL, a concrete verification-oriented language with precise formal syntax and semantics suitable for modelling and verification of component-based systems with ensembles. The syntax of DCCL has been given on two levels—abstract syntax that formally describes the entities that interact in a model, and a concrete syntax that is modelling language describing these entities in a C++-like fashion. We have further presented our verification tool that takes DCCL models as inputs together with LTL property specifications and provides answers about the validity of the properties as well as counterexamples. The tool is based on the DiVinE model-checking environment that uses automata-based LTL model checking algorithms. The interoperation with DiVinE is made possible using a binary interface called CESMI. Finally, we have demonstrated some of the language's and the tool's capabilities on two toy examples. We have discussed the modelling choices that have been made and presented the verification results that have been achieved using our tool.

As for future development of DCCL and the tool, we want to focus on two areas. One is that of further enhancing the language. One idea might be to only allow some parts of

Table 1: Autonomous cars example results.

model	# cars	t.u.	property	result	# states	# transitions	time
Carplan	4	fix	—	—	2 693 382	3 547 940	16.36 s
Carplan	4	fix	`G(!crash)`	yes	2 693 382	3 547 940	19.96 s
Carplan	4	fix	`GF(car1wp1)`	no	5 246 825	10 259 194	129.65 s
Carplan	4	7	—	—	6 100 499	6 707 969	20.64 s
Carplan	4	7	`G(!crash)`	yes	6 100 499	6 707 969	22.36 s
Carplan	4	7	`GF(car1wp1)`	no	11 867 633	19 370 605	163.89 s
Carplan	4	6	—	—	5 600 911	6 212 034	30.50 s
Carplan	4	6	`G(!crash)`	no	11 145 385	12 360 755	63.41 s
Carplan	3	6	—	—	167 204	175 529	0.97 s
Carplan	3	6	`G(!crash)`	yes	167 204	175 529	1.24 s
Carplan	3	6	`GF(car1wp1)`	yes	325 059	506 316	6.39 s
Pedestrian	2	4	—	—	677 620	824 747	3.97 s
Pedestrian	2	4	`G(!crash)`	yes	677 620	824 747	5.04 s
Pedestrian	2	4	`GF(car1wp1)`	no	1 317 330	2 378 705	32.58 s

Table 2: Burglar example results.

size	# guards	t.u.	property	result	# states	# transitions	time
8x8	9	fix	—	—	53 136	117 456	0.45 s
8x8	9	1	—	—	70 848	135 168	0.55 s
8x8	9	1	`F(seen) || G(!centre)`	no	72 712	109 876	1.76 s
8x8	10	1	`F(seen) || G(!centre)`	yes	7 364	11 134	0.04 s
10x10	14	fix	—	—	807 840	1 804 320	7.51 s
10x10	14	1	—	—	1 077 120	2 073 600	8.52 s
10x10	14	1	`F(seen) || G(!centre)`	no	400 749	581 388	16.31 s

the knowledge to be changed by the ensemble communication, leaving others (such as the position of a car in the autonomous cars example) protected. Other ideas include extending the language with time constrains and stochastic behaviour, allowing the user to express probabilities.

The other area is that of extending the verification tool. In order to reduce the state space that has to be explored in the verification phase, we plan to include some of the well known reduction techniques such as symmetry reduction, partial order reduction, etc.

7. REFERENCES

[1] ASCENS. http://www.ascens-ist.eu/.

[2] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *HiBi/PDMC 2010*, pages 4–7. IEEE, 2010.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.

[4] Rocco De Nicola, Gian Luigi Ferrari, Michele Loreti, and Rosario Pugliese. A language-based approach to autonomic computing. In Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer, and Marcello M. Bonsangue, editors, *FMCO*, volume 7542 of *Lecture Notes in Computer Science*, pages 25–48. Springer, 2011.

[5] DiVinE. http://divine.fi.muni.cz/.

[6] George T. Heineman and William T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[7] Matthias M. Hölzl, Axel Rauschmayer, and Martin Wirsing. Software engineering for ensembles. In Martin Wirsing, Jean-Pierre Banâtre, Matthias M. Hölzl, and Axel Rauschmayer, editors, *Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *Lecture Notes in Computer Science*, pages 45–63. Springer, 2008.

[8] Matthias M. Hölzl and Martin Wirsing. Towards a system model for ensembles. In Gul Agha, Olivier Danvy, and José Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 241–261. Springer, 2011.

[9] DCCL Homepage. http://paradise.fi.muni.cz/dccl/.

[10] InterLink. http://interlink.ics.forth.gr/.

[11] Jaroslav Keznikl, Tomás Bures, Frantisek Plášil, and Michal Kit. Towards Dependable Emergent Ensembles of Components: The DEECo Component Model. In *WICSA/ECSA 2012*, pages 249–252. IEEE, 2012.

[12] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[13] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[14] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings, Symposium on Logic in Computer Science (LICS'86)*, pages 332–344. IEEE Computer Society, 1986.