The HIVE Tool for Informed Swarm State Space Exploration

Anton Wijs*

Eindhoven University of Technology, 5612 AZ Eindhoven, The Netherlands

A.J.Wijs@tue.nl

Swarm verification and parallel randomised depth-first search are very effective parallel techniques to hunt bugs in large state spaces. In case bugs are absent, however, scalability of the parallelisation is completely lost. In recent work, we proposed a mechanism to inform the workers which parts of the state space to explore. This mechanism is compatible with any action-based formalism, where a state space can be represented by a labelled transition system. With this extension, each worker can be strictly bounded to explore only a small fraction of the state space at a time. In this paper, we present the HIVE tool together with two search algorithms which were added to the LTSMIN tool suite to both perform a preprocessing step, and execute a bounded worker search. The new tool is used to coordinate informed swarm explorations, and the two new LTSMIN algorithms are employed for preprocessing a model and performing the individual searches.

1 Introduction

In explicit-state model checking (MC), it is checked whether a given system specification yields a given temporal property. This is done by exploring the so-called *state space* of the specification, which is a directed graph describing explicitly all potential behaviour of the system. Since state space exploration algorithms often need to keep track of all explored states in order to efficiently perform the MC task,[1] and since state spaces can be very large, for many years, the amount of available memory in a computer has been the most important bottleneck for MC.

In recent years, however, the increase of available memory in state-of-the-art computers has continued to follow Moore's Law [12], while the increase of their processors' speed no longer has. For MC, this means that large state spaces can be stored in memory, but the time needed to explore them is impractically long, hence a *time explosion problem* has emerged. This can be mitigated by developing *distributed* exploration algorithms, in which a number of computers in a cluster or grid are used to perform an exploration. Many of those algorithms use a partitioning function to assign states to workers, and require frequent synchronisation between these workers, see e.g. [1, 2, 4, 7, 10].

Swarm verification [9] (SV) (and *parallel randomised Depth-First Search* [5]) are recent techniques to perform state space exploration in a so-called *embarrassingly parallel* [6] way, where the individual workers never need to synchronise with each other. In SV, each worker starts at the initial state and performs a search based on Depth-First Search (DFS). The direction of a worker is determined by a given successor ordering strategy. As the direction of a DFS depends on the fact that a stack is used to order successor states (i.e. a Last-In-First-Out strategy), changing this ordering directly influences the direction of the search. By providing each worker a unique strategy, they will explore different parts of the state space first. With this method, some states may be explored multiple times by different workers,

*Supported by the Netherlands Organisation for Scientific Research (NWO) project 612.063.816 *Efficient Multi-Core Model Checking.*

[1] A Depth-First Search can in principle be performed by just using a stack, but this means that the MC task can often not be performed in linear time (depending on the structure of the state space).

Jiri Barnat and Keijo Heljanko (Eds.)
10th International Workshop on
Parallel and Distributed Methods in verifiCation (PDMC 2011)
EPTCS 72, 2011, pp. 91–98, doi:10.4204/EPTCS.72.10

Table 1: The four major functionalities of ISV

LTSMIN	**HIVE**
P1. Trace-counting DFS: Constructs $\mathcal{P}'$ with $tc(s) = \min(1, \Sigma_{s' \in \mathcal{N}'} tc(s'))$.	**F1. Trace selection**: select a swarm trace σ for worker
F2. Informed Swarm Search (ISS): Search of $\mathcal{P}$ restricted to σ	**F3. Update swarm set:** remove inspected traces

but if the property does not hold, any bug states present are likely to be detected very quickly, due to the diversity of the searches, which often means that the whole state space does not have to be explored.

However, if a property holds, each worker will exhaustively explore the whole reachable state space, which means that the benefits of parallelisation are completely lost. Recently, we proposed a mechanism to bound each worker to a particular reachable strict subset of the set of reachable states, in such a way that together, the workers explore the whole state space [14]. This mechanism is compatible with any action-based formalism such as μCRL [8], where each transition in a state space is labelled with some action name corresponding with system behaviour. In this paper, we explain how the *Heuristics Instructor for parallel VErification* (HIVE) tool, which resulted from [14], works in practice. Section 2 presents the functionality of the HIVE tool together with some new algorithms implemented in the LTSMIN tool suite [4]. How all these have been implemented and how the resulting tools can be used is explained in Section 3. In Section 4, experimental results are discussed. Finally, conclusions and pointers for possible future work are given in Section 5.

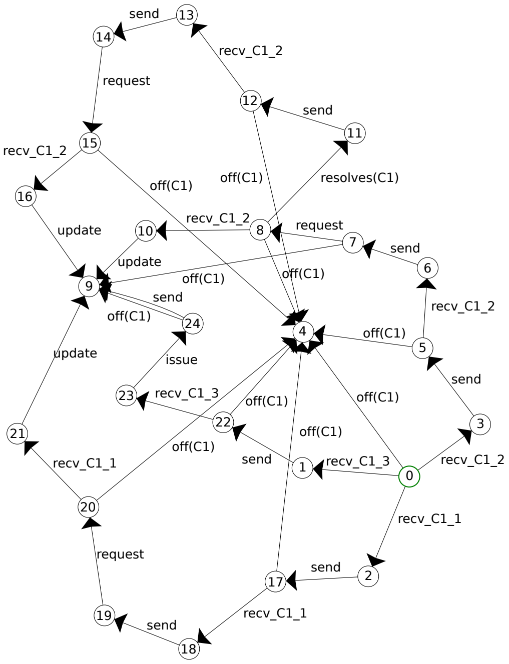
2 The Informed Swarm Exploration Technique

The Setting The so-called *Informed* SV technique (ISV) implemented in HIVE and LTSMIN is applicable if three conditions are met: (1) A system specification $\mathfrak{P}$ should be an implicit description of a *Labelled Transition System* (LTS) $\mathcal{P}$. An LTS $\mathcal{P}$ is a quadruple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, s_{in})$, where s_{in} is the initial state, $\mathcal{S}$ is the set of states reachable from s_{in}, $\mathcal{A}$ is a set of transition labels (actions), and $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the set of transitions between states. With $s \xrightarrow{A} t$, $A \subseteq \mathcal{A}$, we say that there exists an $\ell \in A$ such that $(s, \ell, t) \in \mathcal{T}$. The reflexive transitive closure of $\longrightarrow$ is denoted as $\longrightarrow_*$. In on-the-fly state space exploration, s_{in} and $\mathcal{A}$ are known a priori, but $\mathcal{S}$ and $\mathcal{T}$ are not, and a next-state function $\mathcal{N} : \mathcal{S} \to 2^{\mathcal{S}}$ provides the set of successors of a given state. A state t is the successor of a state s iff $s \xrightarrow{\mathcal{A}} t$. $\mathcal{N}$ is used to construct $\mathcal{S}$ and $\mathcal{T}$, starting at s_{in}. In the following, we use the notation $\mathcal{N} \mid A$, with $A \subseteq \mathcal{A}$, to denote $\mathcal{N}$ restricted to a set of transition labels A, i.e. $\mathcal{N} \mid A(s) = \{s' \in \mathcal{S} \mid \exists \ell \in A.(s, \ell, s') \in \mathcal{T}\}$. Clearly, $\mathcal{N} \mid \mathcal{A} = \mathcal{N}$. Finally, a sequence of actions $\langle \ell_0, \ell_1, \ldots \rangle$ describes all transition sequences (traces) through an LTS $\mathcal{P}$ with $s_{in} \xrightarrow{\{\ell_0\}} s_0 \xrightarrow{\{\ell_1\}} s_1 \cdots$ for some $s_0, s_1, \ldots$. If $\mathcal{P}$ is label-deterministic, i.e. for all $s, t \in \mathcal{S}, \ell \in \mathcal{A}$ with $s \xrightarrow{\ell} t$, there does not exist a state $t' \neq t$ with $s \xrightarrow{\ell} t'$, such an action sequence corresponds to a single trace. Here, we assume that all LTSs are label-deterministic. If this is not the case, relabelling of some transitions can resolve this.

(2) $\mathfrak{P}$ should consist of a *finite number $n > 1$ of process descriptions (e.g. process algebraic terms) in parallel composition*. This is the case for any concurrent system. (3) At least some of these processes in parallel composition, i.e. a subsystem, should yield *finite behaviour*, hence only finite traces. This is not a strict requirement, but if it is not met, then the method relies on bounded analysis of the subsystem, and it does not automatically guarantee anymore that all reachable states are visited.

ISV Say that a specification describes a system of concurrent processes $\mathfrak{P} = \{P_0, \ldots, P_n\}$, with $n \in \mathbb{N}$. ISV exploits the fact that parallel composition is a major cause for state space explosion, and that LTS

$\mathcal{P}$ of $\mathfrak{P}$ is the synchronous product of Ltss $\mathcal{P}_i = (\mathcal{S}^i, \mathcal{A}^i, \mathcal{T}^i, s^i_{in})$ of the P_i ($0 \leq i \leq n$), restricted by some synchronisation rules between processes, given by a symmetric function $\mathfrak{C}$. E.g. $\mathfrak{C}(\ell, \ell') = \ell''$ states that if actions ℓ and ℓ' can be performed by different processes, then the result is action ℓ'' in the system. For the formal details, see [14]. We assume that the $\mathcal{A}^i$ are disjoint (if this is not the case, then some rewriting can resolve this)[2] and that no action is involved in more than one rule defined by $\mathfrak{C}$ (either as an input, or as a result). All this implies that for any $\ell \in \mathcal{A}$, it can be determined whether or not it stems from some behaviour of a particular process P_i. Say that $\mathcal{A}_c \subseteq \mathcal{A}$ is the set of actions stemming from synchronisations, and that $\mathcal{A}^i_c \subseteq \mathcal{A}^i$ is the set of actions of $\mathcal{P}_i$ which are forced to synchronise with other actions, then $\mathcal{A} = (\bigcup_{i \leq n} \mathcal{A}^i \setminus \mathcal{A}^i_c) \cup \mathcal{A}_c$. Now, for any $A \subseteq \mathcal{A}$, we can define $\mathfrak{M}(A)$ as $\{\ell'' \in \mathcal{A}_c \mid \exists \ell \in A, \ell' \notin A . \mathfrak{C}(\ell, \ell') = \ell''\}$, which is the set of actions resulting from synchronisation involving one action in A. Finally, the assumptions about $\mathfrak{C}$ allow us to define a relabelling function $\mathfrak{R}$ as follows: $\mathfrak{R}(\{\ell\}) = \{\ell''\}$ iff there exists an ℓ' such that $\mathfrak{C}(\ell, \ell') = \ell''$, and $\mathfrak{R}(\{\ell\}) = \{\ell\}$, otherwise.



Trace counts	
State	*tc*
0	14
1	3
2	3
3	7
4	1
5	7
6	6
7	6
8	5
9	1
10	1
11	3
12	3
13	2
…	…
24	2

Trace #10
$0 \rightarrow [0, 14\rangle$
$3 \rightarrow [6, 13\rangle$
$5 \rightarrow [6, 13\rangle$
$6 \rightarrow [7, 13\rangle$
$7 \rightarrow [7, 13\rangle$
$8 \rightarrow [7, 12\rangle$
$11 \rightarrow [9, 12\rangle$
$12 \rightarrow [9, 12\rangle$
$13 \rightarrow [10, 12\rangle$
$14 \rightarrow [10, 12\rangle$
$15 \rightarrow [10, 12\rangle$
$4 \rightarrow [10, 11\rangle$

Figure 1: The Lts of an iPod process, with weights, and trace nr. 10

Four basic functionalities are required to perform ISV in practice. These are listed in Table 1; a preprocessing step (P1) involving the analysis of a defined subsystem yielding finite behaviour, and three techniques for the three major phases of ISV (F1-3). P1 and F2 require two new search algorithms, which have been implemented in LTSmin. F1 and F3 have been implemented in the new stand-alone Hive tool. The general procedure to perform an ISV is as follows: First, the user selects a strict subsystem $\mathfrak{P}' \subset \mathfrak{P}$ which is guaranteed to yield finite behaviour (again, alternatively, this behaviour is bounded in the ISV, but then, the overall search could be non-exhaustive). In P1, the Lts $\mathcal{P}' = (\mathcal{S}', \mathcal{A}', \mathcal{T}', s'_{in})$, described by $\mathfrak{P}'$, is constructed and saved to disk, together with a weight function $tc : \mathcal{S}' \rightarrow \mathbb{N}$. For this, we have extended the Dfs implementation in LTSmin, as described in Alg. 1. The tc function (see also Table 1) assigns 1 to deadlock states, i.e. if $\mathcal{N}'(s) = \emptyset$, and the sum of all the successor weights to any other state (note that $\mathcal{N}'$ is the next-state function of $\mathcal{P}'$).

This allows efficient reasoning about the traces through $\mathcal{P}'$; the number of traces represented by a trace prefix from s_{in} to some $s \in \mathcal{S}'$ equals $tc(s)$. E.g., Fig. 1 shows a simplified acyclic Lts[3] of an iPod process as part of the DRM protocol specification from [13], with part of the definition of tc. With this, if we sort the states based on their numbering (which was assigned by the Dfs), then each trace can be uniquely referred to with a natural number: note that the number of possible traces is 14, which

[2] Strictly speaking, a weaker requirement suffices [14].

[3] The actual Lts of this example, in which the actions are extended with some data parameters, consists of 547 states.

corresponds with $tc(0)$. Say that we want to identify trace 10 (shown in Fig. 1). State 0 has successors $\{1,\ldots,4\}$. Sorted by increasing state number, we first consider state 1; since $tc(1) = 3$, we conclude that trace $\langle \mathsf{recv\_C1\_3} \rangle$ represents traces 0 to 2, i.e. 3 traces, starting at trace 0.

Algorithm 1 Trace-counting DFS

```
Require: $\mathfrak{P}' \subset \mathfrak{P}$, $s'_{in}$, $\mathcal{A}'$
Ensure: $\mathcal{P}'$ and $tc : \mathcal{S}' \to \mathbb{N}$ are constructed
  $Closed \leftarrow \emptyset$
  $tc(s_{in}) \leftarrow dfs(s_{in})$
dfs(s) =
  if $s \notin Closed$ then
    $tc(s) \leftarrow 0$
    for all $s' \in \mathcal{N}'(s)$ do
      $tc(s) \leftarrow tc(s) + dfs(s')$
    if $\mathcal{N}'(s) = \emptyset$ then
      $tc(s) \leftarrow 1$
    $Closed \leftarrow Closed \cup \{s\}$
  return $tc(s)$
```

We also have $tc(2) = 3$, therefore $\langle \mathsf{recv\_C1\_1} \rangle$ represents traces 3 to 5. Similarly, $\langle \mathsf{recv\_C1\_2} \rangle$ represents traces 6 to 12, and $\langle \mathsf{off(C1)} \rangle$ represents trace 13. This means that $\langle \mathsf{recv\_C1\_2} \rangle$ is a prefix of trace 10. Since state 3 only has state 5 as a successor, clearly $\langle \mathsf{recv\_C1\_2}, \mathsf{send} \rangle$ is also a prefix (this agrees with $tc(5) = 7$: all 7 traces represented by $\langle \mathsf{recv\_C1\_2} \rangle$ are also represented by $\langle \mathsf{recv\_C1\_2}, \mathsf{send} \rangle$). In this fashion, the complete trace can be constructed following the states listed on the right of Fig. 1. This principle is used for trace selection in HIVE (F1). In ISV, each worker is bounded by a trace through $\mathcal{P}'$, given by HIVE (this will be explained next). Therefore, each trace represents a worker job to be performed, and $\mathcal{P}'$ represents the set of jobs. From a trace $\langle \ell_0, \ldots, \ell_n \rangle$ through $\mathcal{P}'$ ($n \in \mathbb{N}$), a so-called *swarm trace* $\sigma = \langle \mathfrak{R}(\ell_0), \ldots, \mathfrak{R}(\ell_n) \rangle$ can be constructed, taking into account synchronisations with $\mathfrak{P} \setminus \mathfrak{P}'$. Whenever a worker thread can be launched, HIVE selects a swarm trace. When the HIVE tool is launched to start an ISV, this is done first.

A launched worker thread performs an informed swarm search (ISS), implemented in LTSMIN (Alg. 2 and F2). In Alg. 2, σ is the swarm trace assigned by the HIVE tool, and $\sigma(i)$ is the singleton set containing the $(i+1)^{th}$ element of σ (If σ contains fewer than $i+1$ elements, we say that $\sigma(i) = \emptyset$). In the ISS, $\mathcal{P}$ is explored, but not exhaustively: the potential behaviour of the subsystem $\mathfrak{P}'$ is restricted to σ, which restricts exploration of $\mathcal{P}$. For each visited state s, *Next* is extended with $\mathcal{N} \mid (\mathcal{A} \setminus A)(s)$, i.e. all successor states reachable via behaviour of $\mathfrak{P} \setminus \mathfrak{P}'$, and *Step* is extended with $\mathcal{N} \mid \sigma(i)(s)$, i.e. all successor states reachable via the current behaviour in σ.

Algorithm 2 BFS-based ISS

```
Require: $\mathfrak{P}$, $s_{in}$, $\mathcal{A}$, $A = \mathcal{A}' \cup \mathfrak{M}(\mathcal{A}')$, $\sigma$
Ensure: $\mathcal{P}$ restricted to $\sigma$ is explored
  $i \leftarrow 0$
  $Open \leftarrow s_{in}$; $Closed, Next, Step, F_i \leftarrow \emptyset$
  while $Open \neq \emptyset \vee Step \neq \emptyset$ do
    if $Open = \emptyset$ then
      $i \leftarrow i + 1$
      $Open \leftarrow Step \setminus Closed$; $Step, F_i \leftarrow \emptyset$
    for all $s \in Open$ do
      $Next \leftarrow Next \cup \mathcal{N} \mid (\mathcal{A} \setminus A)(s)$
      $Step \leftarrow Step \cup \mathcal{N} \mid \sigma(i)(s)$
      $F_i \leftarrow F_i \cup \{\ell \in A \mid \mathcal{N} \mid \{\ell\}(s) \neq \emptyset\}$
    $Closed \leftarrow Closed \cup Open$
    $Open \leftarrow Next \setminus Closed$; $Next \leftarrow \emptyset$
```

When all states in *Open* are explored, the contents of *Next* is moved to *Open*, after duplicate detection (for which the search history *Closed* is used). Note that when all reachable states have been explored in this manner, i is increased, by which the ISS moves to the next step in σ, and new states become available. The main idea of ISV is to construct the set of all possible traces through $\mathcal{P}'$, and to perform an ISS through $\mathcal{P}$ for each of those traces. This means that eventually $\mathcal{P}$ is completely explored. A proof of correctness can be sketched as follows: say that all traces through $\mathcal{P}'$ have been used by workers to explore $\mathcal{P}$, and that after this, some reachable state $s \in \mathcal{S}$ has never been visited. We will show that this leads to a contradiction. It follows from Alg. 2 that for each state t to be explored, all new states $t' \in \mathcal{N} \mid (\mathcal{A} \setminus A)(t)$ are going to be explored as well, and for some i, $t'' \in \mathcal{N} \mid \sigma(i)(t)$ is going to be added to *Step*. This implies that all states $\hat{t} \in \mathcal{N} \mid (A \setminus \sigma(i))(t)$ are going to be ignored. From this and the fact that s_{in} is explored, it follows that a state s is ignored iff for all traces through $\mathcal{P}$ from s_{in} to s, there exist $t, \hat{t} \in \mathcal{S}$ such that $s_{in} \xrightarrow{\mathcal{A}}_* t \xrightarrow{\ell} \hat{t} \xrightarrow{\mathcal{A}}_* s$, with $\ell \in A \setminus \sigma(i)$, i being the current position in σ when exploring t. Let us consider one of those traces. We call σ' the swarm trace followed to reach t from s_{in} over that trace. Note that this is a prefix of σ. Let us assume that by following σ' extended with ℓ, s can be reached from s_{in}.[4] Since σ' has been derived from a trace through $\mathcal{P}'$ and $\ell \in A$, the extended trace must also be derivable from a trace through $\mathcal{P}'$. But then, since all traces through $\mathcal{P}'$ have been used in the ISV, s must have been visited by some other worker that followed σ', and we have a contradiction.

In case $\mathfrak{P}$ and $\mathfrak{P}'$ sometimes synchronise, the trace counting will produce an over-approximation of the possible set of traces of $\mathfrak{P}'$ *in the context of* $\mathfrak{P}$. This is because in the trace counting, it is always assumed that whenever $\mathfrak{P}'$ needs to synchronise, this can happen in $\mathfrak{P}$. The result is that some swarm traces may not correspond with actual potential behaviour in $\mathcal{P}$. To deal with this, ISS includes a feedback procedure: For every position i in σ, it is recorded in F_i which potential behaviour of $\mathfrak{P}'$ has actually been observed. When finished, ISS returns the F_i, and using these, HIVE can prune away both σ and other, invalid, traces (F3). Since each trace prefix represents a set of traces with *consecutive numbers* (see e.g. the ranges for states in trace 10 in Fig. 1), the set of explored and pruned swarm traces can be represented in a relatively small list of ranges. Initially, the set of swarm traces is empty. Say we explore the LTS $\mathcal{P}$ of the DRM specification, and $\mathcal{P}'$ is as displayed in Fig. 1, and say it is detected that synchronisation with recv_C1_2 at state 0 (to state 3) can actually not happen in $\mathcal{P}$. As already mentioned, $\langle$recv_C1_2$\rangle$ represents $[6,13\rangle$. So after pruning, $[6,13\rangle$ represents the new set of explored traces. Furthermore, elements in the list can often be merged. E.g., if ranges $[0,5\rangle$ and $[8,14\rangle$ have been explored earlier, and range $[5,8\rangle$ is to be added, the result can again be described using a single range $[0,14\rangle$. At all times, HIVE is ready to launch another worker (F1) and to prune more traces (F3). When there are no more swarm traces left to explore, the ISV is finished.

3 Implementation and Using the Tools

Implementation The trace-counting DFS and ISS have both been implemented in an unofficial extension of the LTSMIN toolset version 1.6-19, which has been written in *C*. Since LTSMIN already contains a whole range of exploration algorithms (both explicit-state and symbolic), there was no need to implement new data structures. ISV is very light-weight in terms of communication between the HIVE and the workers, the only information sent to launch an ISS being a swarm trace in the form of a list of actions. This list is being stored in LTSMIN in a linked list, and a pointer traverses this list when exploring, to keep track of the current swarm trace position. In addition to this, a bit set is used to keep track of the encountered actions stemming from $\mathfrak{P}'$ since the last move along the swarm trace. This is done to construct the F_i. A bit set implementation using a tree data structure is available in the LTSMIN toolset.

Unfortunately, it is currently not possible to automatically extract $\mathfrak{P}'$ of a given subsystem from $\mathfrak{P}$, meaning that $\mathfrak{P}'$ must manually be derived from $\mathfrak{P}$. At times, this requires quite some inside knowledge of the description, therefore it is at the moment the main reason that we have not yet performed more experiments. Automatic construction of the $\mathfrak{P}'$ description is listed as future work (see Section 5).

The HIVE tool consists of about 1,200 lines of *C*-code. Because of the communication being light-weight, and because interactions between the workers and HIVE either involve asking for a new trace and receiving it, or sending the results of an ISS, we decided to implement all communication in the request-response method using TCP/IP sockets. During an ISV, HIVE frequently needs to extract traces from the $\mathcal{P}'$, which is kept in memory together with the *tc*-function. Besides this, a linked list L of nodes containing trace ranges (the $[i,j\rangle$ mentioned in Section 2) is maintained, representing the set of explored

[4] This is not true if there are multiple transitions stemming from $\mathfrak{P}'$ on the trace to s not agreeing with σ, but then, we can repeat the reasoning in the proof sketch until there is only one left.

traces. Currently, when a trace is selected (F1), an ID is chosen randomly from L, but one can imagine other selection strategies (see Section 5). Then, the corresponding trace is extracted from $\mathcal{P}'$.

When launching many workers, frequent requests to Hive are to be expected. Therefore, Hive has been implemented with *pthreads*; whenever a worker sends a request, a new thread is launched in Hive to handle the request. If a new swarm trace is required, F1 is performed, and if feedback is given, F3 is performed. The Lts $\mathcal{P}'$ is never changed, hence no race conditions can occur when multiple threads read it, but L is frequently accessed and updated, when selecting a trace and processing feedback, the latter involving writing. For this reason, we introduced a data lock on L. We plan to use more fine-grained locking in the future, but we have not yet experienced a real slowdown when using one lock.

During an ISV, Hive keeps accepting new requests until L has one node containing the range $[0, tc(s'_{in}))$. From that moment on, any requests are answered with the command to terminate, effectively ending all worker executions. The same is done if a worker reports in its feedback that a counter-example to a property to check has been found, because the ISV can stop immediately in that case.

Finally, all has been tested on Linux (Red Hat 4.3.2-7 and Debian 6.0.1) and Mac OS X 10.6.8.

Setting up and launching an ISV In the following, we assume that we have a μCRL specification named spec.mcrl describing $\mathfrak{P}$, and a specification named specsub.mcrl describing $\mathfrak{P}'$. Actually, any action-based modelling language compatible with LTSmin is suitable for ISV as well. A μCRL specification is usually first linearised to a tbf file, using the μCRL toolset [3], which is subsequently used as the actual input of LTSmin. Having spec.tbf and specsub.tbf, the weighted $\mathcal{P}'$ is saved to disk as follows, with ⟨sub⟩ being the chosen base name for the files storing the weighted $\mathcal{P}'$:[5]

```
lpo2lts-grey –getswarm=⟨sub⟩ specsub.tbf
```

This produces the files sub.swh, sub.swc, and sub.sww, containing the actions in $\mathcal{A}'$, the transitions in $\mathcal{T}'$ (with actions and states represented by numbers), and the weights of the states, respectively. Actually, if specsub.tbf yields infinite behaviour, this can be bounded by a depth n using the option –swbound=⟨n⟩.

The Hive can now be launched on the same machine by invoking the following, with ⟨portnr⟩ being the port number it is supposed to listen at for incoming requests:

```
hive ⟨portnr⟩ ⟨sub⟩
```

An Iss can be started as follows, ⟨server⟩ being the IP address of the machine running Hive:

```
lpo2lts-grey –swarm=⟨sub⟩ –hiveserver=⟨server⟩ –hiveport=⟨port⟩ spec.tbf
```

Note that each Iss also needs information on $\mathcal{P}'$. Actually, only sub.swh is read from disk, to learn $\mathcal{A}'$. Therefore, this file should be available on all machines where Iss*s* are started. Finally, in practice, one often wants to start many Isss simultaneously, and start a new Iss every time one terminates. This whole procedure can be launched using the shell script hive_launch.sh.

4 Experimental Results

Table 2 shows experimental results using the μCRL [8] specifications of a DRM procotol [13] and the Link Layer Protocol of the IEEE-1394 Serial Bus (Firewire) [11]. We were not yet able to perform

[5] For μCRL specifications, lpo2lts-grey is the explicit-state space generator of LTSmin. For other modelling languages, another appropriate LTSmin tool should be used.

Table 2: Results for bug-free cases with SV and ISV, 10 and 100 workers.

case	# workers	search	results					
			# est. runs	*# runs*	*max. # states*	*max. time*	*total # states*	*total time*
DRM (1nnc, 3ttp)	10	SV	10	10	13,246,976	19,477 s	132,469,760	19,477 s
	10	ISV, 1 iPod	5,124	45	2,352,315	2,832 s	85,966,540	14,157 s
	10	ISV, 2 iPods	$1.31 * 10^{13}$	7,070	70,211	177 s	353,591,910	125,139 s
	100	ISV, 2 iPods	$1.31 * 10^{13}$	9,900	70,211	175 s	361,050,900	17,325 s
1394 (3 link ent.)	10	SV	10	10	137,935,402	105,020 s	1,379,354,020	105,020 s
	10	ISV	$3.01 * 10^{9}$	1,160	236,823	524 s	235,114,520	60,784 s
	100	ISV	$3.01 * 10^{9}$	1,400	236,823	521 s	252,206,430	7,294 s

# est. runs: estimated # runs needed (# of swarm traces for ISV). # runs: actual # runs needed. *total (max.) # states*: total (largest) # states explored (in a single search). *total (max.) time*: total (longest) running time (of a single search).

more experiments using other specifications, mainly because subsystem specifications still need to be constructed manually, which requires a deep understanding of the system specifications. The experiments were performed on a machine with two dual-core AMD OPTERON (tm) processors 885 2.6 GHz, 126 GB RAM, running Red Hat 4.3.2-7. We simulated the presence of 10 and 100 workers for each experiment (the fully independent worker threads can also be run in sequence). This has some effect on the results: in order to simulate 10 and 100 parallel ISSs, Hive postponed the processing of Iss feedback until 10 and 100 of them had been accumulated, respectively. When the ISSs truly run in parallel, this feedback processing is done continuously, and redundant work can therefore be avoided at an earlier stage. In the DRM case, we selected both one and two iPod processes for $\mathfrak{P}'$, and in the Firewire case, a bounded analysis of one of the link protocol entities resulted in $\mathcal{P}'$. The SV runs have been performed with the Dfs of LTSmin. Since the specifications are correct, there is no early termination for the explorations, meaning that in SV, all reachable states are explored 10 times. In the DRM case, ISV based on one iPod process leads to an initial swarm set with 5,124 traces, 45 of which were actually needed for different runs. Each run needed to explore *at most* 18% of $\mathcal{P}$, and in total, the number of states explored was smaller than in the SV. ISV based on two iPod processes leads to a much larger swarm set, and clearly, feedback information is essential. ISV with 10 parallel workers explored in total 2.5 times more states than SV, but each Iss covered at most $\frac{1}{2}\%$ of $\mathcal{P}$, meaning that they needed a small amount of memory. This demonstrates that ISV is useful in a network where the machines do not have large amounts of RAM. In the Firewire case with 10 parallel workers, each Iss explored at most only $\frac{1}{6}\%$ of $\mathcal{P}$, and in total, the SV explored 83% more states than the ISV. In terms of scalability related to the number of parallel workers, the results with 100 workers show that the overall execution times can be drastically reduced when increasing the number of workers: compared to having 10 workers, 100 workers reduce the time by 86% in the DRM case, and 88% in the Firewire case. The number of ISSs has actually increased, but we expect this to be an effect of the simulations of parallel workers, as explained before.

A full experimental analysis of the algorithms would also incorporate cases with bugs, to test the speed of detection. This is future work, but since ISV has practically no overhead compared to SV, and the ISSs are embarrassingly parallel and explore very different parts of a state space, we expect ISV and SV to be comparable in their bug-hunting capabilities. Finally, we chose not to compare ISV experimentally with other distributed techniques (e.g. those using frequent synchronisations), because there are too many undesired factors playing a role when doing that (e.g. implementation language, modelling language, level of expertise of the user with the model checker).

5 Conclusions and Future Work

We presented the functionality of the Hive tool and two new LTSmin algorithms for ISVs. ISV is an SV method for action-based formalisms to bound the embarrassingly parallel workers to different Lts parts. Worst case, if the system under verification is correct, no worker needs to perform an exhaustive exploration, and memory and time requirements for a single worker can remain low.

Tool availability Both the ISV extended version of LTSmin and Hive are available at `http://www.win.tue.nl/~awijs/suppls/hive_ltsmin.html`.

Future work We plan to further develop the Hive tool such that a description $\mathfrak{P}'$ of a given subsystem can automatically be derived from a given description $\mathfrak{P}$. We also wish to investigate which kind of subsystems are particulary effective for the work distribution in ISV, and which are not, so that an automatic subsystem selection method can be derived. If $\mathfrak{P}'$ yields infinite behaviour, we want to support its full behaviour automatically in the future. As long as $\mathcal{P}$ is finite-state, this should be possible. Furthermore, we want to investigate different strategies to select swarm traces and to guide individual ISSs. Finally, we will perform more experiments with much larger state spaces, using a computer cluster.

References

[1] J. Barnat, L. Brim, M. Ceška & P. Ročkai (2010): *DiVinE: Parallel Distributed Model Checker*. In: *HiBi/PDMC'10*, pp. 4–7, doi:10.1109/PDMC-HiBi.2010.9.

[2] B. Bingham, J. Bingham, F.M. de Paula, J. Erickson, G. Singh & M. Reitblatt (2010): *Industrial Strength Distributed Explicit State Model Checking*. In: *HiBi / PDMC 2010*, IEEE, pp. 28–36, doi:10.1109/PDMC-HiBi.2010.13.

[3] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser & J.C. van de Pol (2001): *μCRL: A Toolset for Analysing Algebraic Specifications*. In: *CAV'01*, *LNCS* 2102, Springer, pp. 250–254, doi:10.1007/3-540-44585-4_23.

[4] S.C.C. Blom, J.C. van de Pol & M. Weber (2010): LTSmin*: Distributed and Symbolic Reachability*. In: *CAV'10*, *LNCS* 6174, pp. 354–359, doi:10.1007/978-3-642-14295-6_31.

[5] M.B. Dwyer, S.G. Elbaum, S. Person & R. Purandare (2007): *Parallel Randomized State-space Search*. In: *ICSE'07*, IEEE, pp. 3–12, doi:10.1109/ICSE.2007.62.

[6] I. Foster (1995): *Designing and Building Parallel Programs*. Addison-Wesley.

[7] H. Garavel, R. Mateescu, D. Bergamini, A. Curic, N. Descoubes, C. Joubert, I. Smarandache-Sturm & G. Stragier (2006): *DISTRIBUTOR and BCG_MERGE: Tools for Distributed Explicit State Space Generation*. In: *TACAS'06*, *LNCS* 3920, Springer, pp. 445–449, doi:10.1007/11691372_30.

[8] J.F. Groote & A. Ponse (1995): *The Syntax and Semantics of μCRL*. In: *ACP'94*, Springer, pp. 26–62.

[9] G.J. Holzmann, R. Joshi & A. Groce (2008): *Swarm Verification*. In: *ASE'08*, IEEE, pp. 1–6, doi:10.1109/ASE.2008.9.

[10] F. Lerda & R. Sista (1999): *Distributed-Memory Model Checking with SPIN*. In: *SPIN'99*, *LNCS* 1680, Springer, pp. 22–39, doi:10.1007/3-540-48234-2_3.

[11] S.P. Luttik (1997): *Description and Formal Specification of the Link Layer of P1394*. SEN-R 9706, CWI.

[12] G.E. Moore (1998): *Cramming more Components onto Integrated Circuits*. Proc. of the IEEE 86(1), pp. 82–85, doi:10.1109/JPROC.1998.658762.

[13] M. Torabi Dashti, S. Krishnan Nair & H.L. Jonker (2008): *Nuovo DRM Paradiso: Towards a Verified Fair DRM Scheme*. Fundamenta Informaticae 89(4), pp. 393–417.

[14] A.J. Wijs (2011): *Towards Informed Swarm Verification*. In: *NFM'11*, *LNCS* 6617, Springer, pp. 422–437, doi:10.1007/978-3-642-20398-5_30.