The computational behaviour of the SCIFF abductive proof procedure and the SOCS-SI system

Marco Alberti
ENDIF, Università di Ferrara

Federico Chesani
DEIS, Università di Bologna
The high computational cost of abduction has limited the application of this powerful and expressive formalism to practical cases.

SCIFF is an abductive proof procedure used for verifying the compliance of agent behaviour to interaction protocols in multi-agent systems; SCIFF has been integrated in SOCS-SI, a system able to observe the agent interaction, pass it to SCIFF for the reasoning process and to display in a GUI the results of the SCIFF computation.

In order to assess the applicability of SCIFF and SOCS-SI to practical cases, we have evaluated qualitatively and experimentally (not yet formally) their computational behaviour, as far as limitations and scalability. In this paper we show the results of the analysis.

Keywords: Abduction, Proof Procedure, Experimentation, Agent Interaction Verification

1 Introduction

Abductive Logic Programming [KKT98] has often been proposed, in the last two decades, as a powerful and expressive formalism for hypothetical reasoning. However, its application to concrete cases has always had to face the high computational cost of abductive proof procedures.

SCIFF is an abductive proof procedure developed and implemented in the context of the SOCS [SOC] project in order to verify the compliance to interaction protocols of the agent behaviour in multi-agent systems. SCIFF has been integrated in SOCS-SI, a system which can observe the agent interaction, pass it to SCIFF for the reasoning process and display the results in a GUI.

While no formal results have yet been proved about the computational complexity of SCIFF, it has seemed apparent that the computational cost can greatly vary depending on the abductive programs used for the specification of the interaction protocols. SOCS-SI introduces a further issue, since it keeps track of all the intermediate nodes explored by the SCIFF.

Therefore, the viability of SCIFF and SOCS-SI to verification of practical multi-agent systems has required, as a condition, a qualitative and experimental analysis of their computational behaviour. This paper presents the results of such analysis.

First of all, we have generated particular test instances with the purposes of stressing the proof procedure by varying the parameters in the following way:

- by increasing the depth of the search tree;
- by increasing the breadth of the search tree;

The test instances created did not have any particular meaning or significance, and were created for the specific purpose of achieving the worst scenario for the SCIFF. Each test has been repeated in 2 different settings, i.e., by having the SCIFF executing either alone or in conjunction with the SOCS-SI tool.

Once determined the computational limits of SCIFF and SOCS-SI, we have studied the computational behavior with respect, in particular, to scalability. In this further testing phase, we referred to a real scenario, namely a Combinatorial Auction. Our main motivation is to confront our work with a concrete case of agent interaction, and hopefully to prove the feasibility of using an abductive proof procedure for compliance verification of agent interaction.

The paper is organized as follow: in Section 2 we introduce the SOCS project, with particular attention to the SCIFF proof procedure and SOCS-SI tool developed within the project. In Section 3 we present some naive tests and their results: these tests were
aimed to stress the SCIFF proof procedure by varying the \textit{depth} and the \textit{breadth} of the searching tree. In Section 4 instead we present a test placed in a more real scenario, and aimed to get some qualitative information about a real-life application, combinatorial auctions. Finally, in Section 5, we summarize the results obtained during the tests.

2 The SOCS Project

The IST-2001-32530 (SOCS) project [SOC] investigated the application of Computational Logic techniques to multi-agent systems. In particular, a part of the project was about the specification and verification of agent interaction protocols in an abductive framework [AGL+03], which defines declarative notions of \textit{fulfilment} and \textit{violation}, mapping, respectively, an agent behaviour that is compliant or non compliant to the interaction protocols. In this section, we briefly recall the operational aspect of the framework: an abductive proof procedure, implemented with CL-based tools and integrated in a system equipped with a GUI and able to observe the agent interaction to be checked for compliance.

2.1 The SCIFF Proof Procedure

SCIFF [AGL+04] is an abductive proof procedure which extends the IFF proof procedure by Fung and Kowalski [FK97]. A SCIFF program is composed of $KB_S$, a logic program whose clauses can have abducibles in their body, and $IC_S$, a set of integrity constraints in the form of implications whose heads are disjunctions.

The extensions to the IFF proof procedure stem from the application domain for which SCIFF has been designed (verification of compliance of agent interaction to interaction protocols), which requires the following features:

- universal variables in abducibles;
- dynamic enlargement of the knowledge base with facts;
- constraints (à la Constraint Logic Programming) applied to variables.

Operationally, SCIFF is a rewriting proof procedure which, to each state of the computation (or node), applies one or more transitions to generate children nodes. In this way, starting from an initial node, a tree is generated. In general, some of the leaf nodes will be of success, and others of violation. Success nodes represent the \textit{fulfilment} of the interaction protocols by the agents; failure nodes represent \textit{violation}.

SCIFF has been implemented in SICStus Prolog, exploiting, in particular, its Constraint Handling Rules [Frü98] library. While no formal results have yet been proven about the SCIFF computational complexity, qualitative and experimental analysis suggest that the required time to compute an answer can vary dramatically, depending on several input factors.

Qualitatively, the computational complexity of SCIFF can be evaluated as follows. Each SCIFF computation produces a search tree whose \textit{depth} and \textit{breadth} determine the total number of nodes, and thus the time needed to explore the (whole) tree. As the proof tree is explored by SCIFF in a depth-first fashion, the depth of the tree, together with the size of a single node, also impacts on space requirements. For both time and space, the worst case is when each branch leads to failure, because in this case the whole tree is explored in search of a success node.

Intuitively, the \textit{depth} of the search tree depends on the total number of \textit{events} (the facts added dynamically to the knowledge base).

The \textit{breadth} of the search tree, instead, is influenced by both the number of disjuncts in the head of the SCIFF integrity constraints, and the alternative branches arising in several of the SCIFF transitions. For example, one of the branches generated by the SCIFF transition \textit{fulfilment} (see [AGL+04]) can be safely pruned, provided that the set $IC_S$ respects some syntactic conditions, whose discussion is beyond the scope of this paper. In such cases, it is possible to optimize the performance of SCIFF by reducing the number of generated branches. In this paper, we call this optimized SCIFF behaviour \textit{f-deterministic}, as opposed to the \textit{f-non-deterministic}, which does not perform the pruning.

2.2 The SOCS-SI Tool

The SCIFF implementation has been integrated into SOCS-SI [ACG+04], a Java-based system equipped with a Graphical User Interface, which has been interfaced to platforms for multi-agent systems (PROSOCS [SKL+04], Jade [JAD]), the TuCSoN coordination platform [OZ99], and an email exchange system.

The core of SOCS-SI is composed of three main modules (see Fig. 1), namely:

- Event Recorder: fetches events from different sources and stores them inside the History Manager.
- History Manager: receives events from the Event Recorder and composes them into an “event history”.
- Social Compliance Verifier: fetches events from the History Manager and passes them on to the proof procedure in order to check the compliance of the history to the specification. It receives the
expectations from the proof-procedure and visualises them in the GUI.

The first important assumption we made in our model is that agents communicate by exchanging messages; these messages are the “events” that the S\(\text{C}\)IFF will check for verification. A second important assumption we made is that the communication layer provides means for accessing all the messages exchanged between agents. Hence, the SOCS-SI tool can be aware of the messages exchanged. This is quite a strong assumption, considering the highly distributed characteristic of MAS. However, as already showed in agents platform like JADE [JAD] and PROSOCS [SKL+04], this assumption can be translated in a feasible implementation.

The Event Recorder fetches events and records them into the History Manager, where they become available to the S\(\text{C}\)IFF proof-procedure. As soon as the proof-procedure is ready to process a new event, it fetches one from the History Manager. The event is processed and the results of the computation are returned to the GUI. The proof-procedure then continues its computation by fetching another event if there is any available, otherwise it suspends, waiting for new events.

In order to support different agent platforms (and more general communication layers), a “plug-in-like” mechanism has been implemented in the Event Recorder. Beside the integration with the communication layer developed within the SOCS project, we have developed plug-ins to support JADE [JAD], TuCSoN [ROD02], and the standard e-mail system (human agents communicating by exchanging properly formatted e-mails). For testing and debugging purposes, we also developed modules to interact with the user prompt, as well as with the file system: i.e., it is possible to log to a file agent interactions and analyze it a-posteriori. We will take advantage of particular of this last feature in order to simulate interactions that worse the performances of the S\(\text{C}\)IFF and SOCS-SI tool.

The Graphical User Interface (Figure 2) provides access to the proof state, i.e., the results of the computation, returned by the proof-procedure. These results are expressed in terms of society expectations about the future behavior of agents, and also in terms of fulfilled expectations and violations of social rules. A graphical tree-view of the whole computation is provided: interestingly, the shown tree bears both an operational and a logical interpretation. The operational interpretation is an intuitive graphical form of a log-file, showing the most significant computational steps, useful for debugging purposes. The logical meaning is an or-tree of the possible S\(\text{C}\)IFF derivations timed by the incoming events. For each incoming event that enriches the knowledge base, the frontier of the explored proof-tree (which is a logical disjunction, as in various proof-procedures [FK97]) is shown. The user can inspect each of the nodes, and see in the main window the state of the computation, i.e., the conjunction of logical formulae of the types in the S\(\text{C}\)IFF language: abducibles, constraints, literals, implications. As it is quite easy to guess, the possibility of inspecting the proof state for each node of the exploration tree is quite expansive in term of memory requirements, and it is one of the main reasons for the degradation of performances when S\(\text{C}\)IFF is used on conjunction with SOCS-SI.

3 Tests in simple scenarios

In this section we test the computational time required by S\(\text{C}\)IFF/SOCS-SI to process some simple protocols and histories. In particular, the aim of this group of tests is not to establish absolute values about per-
performances, but rather to understand how the com-
putation time required to provide an answer is af-
fected by changes in the length of the history pro-
cessed (Section 3.1), and by changes in the alterna-
tives dialogues allowed by the protocol (Section 3.2).
The last experiment in particular roughly correspond
to increasing the breadth of the search tree explored
by SIFF, whether the former corresponds to increasing
the depth of the tree.
The output considered is only the computational
time required to elaborate an answer. For test in-
stances of certain dimension (see Tables 1 and 2), it
was not possible to achieve the completion of the test,
mainly for limitations of the hardware. This condition
is expressed by placing a question mark in the results
tables, instead of a value.

All the tests were designed to provide a positive
answer by SIFF, and were executed on a PC with a
2 GHz Pentium IV CPU, 512 MB of RAM, Windows

3.1 Increasing the depth of the exploring
tree

The aim of this test is to evaluate the impact of his-
tories of various length on SIFF and SOCS-SI. In
order to do so, we have considered a very naive proto-
col, presented in the Specification 3.1, along with the
history used to test it. The considered results, pre-
sented in the Table 1, are the time required to elaborate
SOCS-SI and SIFF respectively to elaborate the histories
of various length, and to provide the expected positive
answer. The used protocol does not contain any alter-
native (disjunction) in the head of the rule: each time
an appropriate event is processed, a new expectation
is generated and, if possible, fulfilled. The parameter
varied was the number of messages (events) composing
each history, and results are shown in Table 1.

### Specification 3.1

The naive protocol and an example history used for testing SIFF and SOCS-SI performances with the increasing of the depth of the search tree.

\[
\begin{align*}
\text{H} & \text{tell}(A, B, a\text{Question}(a\text{Parameter}), D), T) \\
\rightarrow & \text{E}((\text{tell}(B, A, a\text{Answer}(a\text{Parameter}), D), T_1) \land T_1 \geq T)
\end{align*}
\]

\[
\begin{align*}
tell & (a, b, a\text{Question}(a\text{Parameter}), d_1, t) \\
tell & (b, a, a\text{Answer}(a\text{Parameter}), d_1, t) \\
tell & (a, b, a\text{Question}(a\text{Parameter}), d_2, t) \\
tell & (b, a, a\text{Answer}(a\text{Parameter}), d_2, t) \\
\cdots \\
tell & (a, b, a\text{Question}(a\text{Parameter}), d_x, t) \\
tell & (b, a, a\text{Answer}(a\text{Parameter}), d_x, t)
\end{align*}
\]

Due to the simplicity of the protocol used in this
test, it would not be correct to assume the results as
meaningful in their absolute values. In fact, although
in this test the SIFF was able to elaborate up to 5000
messages, this does not mean that in a real scenario it
would be able to. The test instead provide two very
useful information about the behavior of SIFF and
SOCS-SI. First of all, the time required to elaborate
longer histories increases in an almost quadratic way,
as it is possible to observe in Figure 3. Secondly, the
SOCS-SI has a big impact on performances, in respect
with SIFF running without a GUI. Not only the
SOCS-SI lowers the maximum number of processable
events before an “Out–of–Memory” error, but also the
performances are worsened, with a factor that is not
constant, but that tends to increase.

![Figure 3: Performances with histories of increasing dimension.](image)

Due to the simplicity of the protocol used in this
test, it would not be correct to assume the results as
meaningful in their absolute values. In fact, although
in this test the SIFF was able to elaborate up to 5000
messages, this does not mean that in a real scenario it
would be able to. The test instead provide two very
useful information about the behavior of SIFF and
SOCS-SI. First of all, the time required to elaborate
longer histories increases in an almost quadratic way,
as it is possible to observe in Figure 3. Secondly, the
SOCS-SI has a big impact on performances, in respect
with SIFF running without a GUI. Not only the
SOCS-SI lowers the maximum number of processable
events before an “Out–of–Memory” error, but also the
performances are worsened, with a factor that is not
constant, but that tends to increase.

<table>
<thead>
<tr>
<th>No. of Messages</th>
<th>SOCS-SI Time(sec.)</th>
<th>SIFF Time(sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.13</td>
<td>0.42</td>
</tr>
<tr>
<td>3</td>
<td>0.18</td>
<td>0.43</td>
</tr>
<tr>
<td>20</td>
<td>0.68</td>
<td>0.51</td>
</tr>
<tr>
<td>30</td>
<td>0.98</td>
<td>0.51</td>
</tr>
<tr>
<td>50</td>
<td>1.82</td>
<td>0.57</td>
</tr>
<tr>
<td>100</td>
<td>6.13</td>
<td>0.76</td>
</tr>
<tr>
<td>150</td>
<td>13.13</td>
<td>0.97</td>
</tr>
<tr>
<td>200</td>
<td>21.68</td>
<td>1.35</td>
</tr>
<tr>
<td>250</td>
<td>33</td>
<td>1.62</td>
</tr>
<tr>
<td>300</td>
<td>47.05</td>
<td>1.99</td>
</tr>
<tr>
<td>400</td>
<td>82.47</td>
<td>3.00</td>
</tr>
<tr>
<td>1000</td>
<td>?</td>
<td>12.16</td>
</tr>
<tr>
<td>2000</td>
<td>?</td>
<td>41.21</td>
</tr>
<tr>
<td>3000</td>
<td>?</td>
<td>92.56</td>
</tr>
<tr>
<td>4000</td>
<td>?</td>
<td>165.89</td>
</tr>
<tr>
<td>5000</td>
<td>?</td>
<td>259.02</td>
</tr>
</tbody>
</table>

Table 1: Testing performances of SIFF and SOCS-SI
while increasing the depth of the search tree.
3.2 Increasing the breadth of the exploration tree

The purpose of this test is to understand how performances of SCIff and SOCS-SI change if the breadth of the search tree increases. To do so, we have developed a different approach w.r.t. the test presented in Section 3.1. In this test, in fact, we vary the protocol definition by increasing the number of disjunction in the head of an Integrity Constraint; the history used, instead, is of a fixed length. The protocol, presented in the Specification 3.2, is again a naive protocol, where the constraint changes for every setting: a subscript $x$ indicates the total number of disjunction. The history has been thought in order to fulfill the protocol w.r.t. the expectation presented in the last disjunction. Since the SCIff explores the search tree by expanding the possible branches following the definition order of the disjunctions in the constraint, this history forces the SCIff proof procedure to explore all the tree. The results obtained are presented in the Table 2.

**Specification 3.2** The naive protocol and an example history used for testing SCIff and SOCS-SI performances with the increasing of the breadth of the search tree.

- **$H(\text{tell}(A, B, a\text{Question}(a\text{Parameter}), D), T)$**
- **$E(\text{tell}(B, A, \text{answer}_1(a\text{Parameter}), D), T_1)$** and $T_1 \geq T$
- **$\lor E(\text{tell}(B, A, \text{answer}_2(a\text{Parameter}), D), T_2)$** and $T_2 \geq T$
- **$\lor E(\text{tell}(B, A, \text{answer}_3(a\text{Parameter}), D), T_3)$** and $T_3 \geq T$
- **$\lor \ldots$**
- **$\lor E(\text{tell}(B, A, \text{answer}_{x-1}(a\text{Parameter}), D), T_{x-1})$** and $T_{x-1} \geq T$
- **$\lor E(\text{tell}(B, A, \text{end}(a\text{Parameter}), D), T_x)$** and $T_x \geq T$

- **tell(a, b, a\text{Question}(a\text{Parameter}), d_1) $d_1$**
- **tell(a, b, \text{end}(a\text{Parameter}), d_1, 2)**
- **close\text{history}.$**

Also for this test, the absolute values are not really meaningful, due to the to simplicity of the protocol used. However, the results show that the time requirements increase with a almost quadratic coefficient w. r. t. the increasing of the disjunctions in the protocol.

Then, in Figure 4 it is possible to appreciate the overhead introduced by the SOCS-SI, suggesting how much the GUI impacts on the overall performances.

We did not test how performances could have changed using the $f$-deterministic version of SCIff.

<table>
<thead>
<tr>
<th>No. of Messages</th>
<th>SOCS-SI Time(sec.)</th>
<th>SCIff Time(sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.032</td>
<td>0.015</td>
</tr>
<tr>
<td>5</td>
<td>0.041</td>
<td>0.016</td>
</tr>
<tr>
<td>10</td>
<td>0.193</td>
<td>0.131</td>
</tr>
<tr>
<td>20</td>
<td>0.219</td>
<td>0.063</td>
</tr>
<tr>
<td>30</td>
<td>0.375</td>
<td>0.109</td>
</tr>
<tr>
<td>40</td>
<td>0.578</td>
<td>0.188</td>
</tr>
<tr>
<td>50</td>
<td>0.859</td>
<td>0.282</td>
</tr>
<tr>
<td>100</td>
<td>2.813</td>
<td>0.921</td>
</tr>
<tr>
<td>200</td>
<td>10.968</td>
<td>3.360</td>
</tr>
<tr>
<td>300</td>
<td>25.423</td>
<td>7.518</td>
</tr>
<tr>
<td>400</td>
<td>46.390</td>
<td>12.937</td>
</tr>
<tr>
<td>500</td>
<td>71.496</td>
<td>19.875</td>
</tr>
<tr>
<td>1000</td>
<td>111.750</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Testing performances of SCIff and SOCS-SI while increasing the depth of the search tree.

![Figure 4: Performances with protocols that allow increasing alternatives.](image)

In fact, in this dummy scenario, it has no sense to introduce knowledge about the domain, and thus it is not possible to take advantage of the $f$-deterministic SCIff.

4 Tests in a real scenario

In this section, we show some experimental results obtained applying SCIff to the verification of compliance to the combinatorial auction protocols described in [ACG+05]. While not being an exhaustive experimentation, the results show the effect on the time costs of SCIff of the breadth and depth of the search tree.

The branching factor of the proof tree obviously impacts heavily on the computational costs of SCIff. In order to show its effect, we have performed some experiments on a combinatorial auction scenario [ACG+05] varying two parameters which contribute to determine the breadth of the proof tree:

1. SCIff version ($f$-non-deterministic vs. $f$-deterministic, see Sect. 2.1);
2. social specification. Spec. 4.1) shows two versions of a social integrity constraint used in the protocol, which are semantically equivalent (i.e., an agent behaviour that respects one will respect the other, and vice-versa), but are verified by $\text{SCIFF}$ in a computationally different way. The first version expresses with a disjunction in the head that the auctioneer can either declare a bid winning (first disjunct) or declare it losing (second disjunct). In the second version, this alternative is expressed by means of a domain variable: intuitively, the auctioneer must declare each bid Answer, where Answer can be either win or lose. Operationally, in the first case, two branches are generated by $\text{SCIFF}$; in the second case, only one branch is generated and the binding of the domain variable is delayed.

In particular, we measure the computation time for sequences of auctions with different numbers of bidders in the two following implementations of the protocol:

1. \textit{f-non-deterministic} $\text{SCIFF}$, protocol with disjunction (which we call the first setup of $\text{SCIFF}$ and protocol);

2. \textit{f-deterministic} $\text{SCIFF}$, protocol with no disjunction (which we call the second setup of $\text{SCIFF}$ and protocol).

The protocol is in both cases the one reported in [ACG+05], apart from the fourth Social Integrity Constraint which, in each setup, is one of those the one in Spec. 4.1: in the first case, the alternative is expressed by means of a disjunction, in the second by means of a variable with domain.

The protocols have been run by varying the number $N$ of bidders, in two different cases.

- In each run of the first case:
  
  1. the auctioneer sends an openauction message to each of the $N$ bidders;
  2. each of the $N$ bidders places a bid;
  3. the auctioneer issues a closeauction message to each of the $N$ bidders;
  4. the auctioneer notifies each of the $N$ bidders with either a win or a lose message, thus resulting in $4N$ total messages exchanged.

- In each run of the second case, the last notification to one of the bidders is missing, thus resulting in a violation of the protocol and $4N - 1$ total messages.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{Bidders} & \textbf{Time(sec.)} & \textbf{Bidders} & \textbf{Time(sec.)} \\
\hline
5 & 1 & 5 & 1 \\
10 & 1 & 10 & 1 \\
15 & 2 & 15 & 2 \\
20 & 3 & 20 & 5 \\
25 & 4 & 25 & 8 \\
30 & 6 & 30 & 10 \\
35 & 9 & 35 & 15 \\
40 & 10 & 40 & 18 \\
45 & 12 & 45 & 23 \\
50 & 21 & 50 & 30 \\
\hline
\end{tabular}
\caption{Combinatorial Auction case 1: Fulfillment}
\end{table}

The experiments were run on a PC with a 2 GHz Pentium IV CPU, 512 MB of RAM, Linux 2.4.18, glibc 2.2.5 and SICStus Prolog 3.10.1. Reported times are in seconds.

In case of fulfillment (see Table 3), the first setup of $\text{SCIFF}$ and protocol seems to scale well with the number of bidders and, in fact, it achieves better execution timing than the second (also shown in Fig. 5). This is basically due to the fact that the chosen setup of interactions directly leads to a successful

![Figure 5: Proof performance on a basic auction (compliant)](image-url)
SCIFF derivation, and only one branch of the tree is explored.

<table>
<thead>
<tr>
<th>Bidders</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>?</td>
</tr>
<tr>
<td>4</td>
<td>55</td>
</tr>
<tr>
<td>5</td>
<td>?</td>
</tr>
<tr>
<td>10</td>
<td>?</td>
</tr>
<tr>
<td>15</td>
<td>?</td>
</tr>
<tr>
<td>20</td>
<td>?</td>
</tr>
<tr>
<td>25</td>
<td>?</td>
</tr>
<tr>
<td>30</td>
<td>?</td>
</tr>
<tr>
<td>35</td>
<td>?</td>
</tr>
<tr>
<td>40</td>
<td>?</td>
</tr>
<tr>
<td>45</td>
<td>?</td>
</tr>
<tr>
<td>50</td>
<td>?</td>
</tr>
</tbody>
</table>

Table 4: Combinatorial Auction case 2: Violation

In the case of violation (see Table 4), however, the first setup of SCIFF and protocol explodes for a very small number of bidders. The experiment with 5 bidders was suspended since this did not reach the answer of violation after several minutes of computing time; no experiments were performed with a higher number of computees, which would have made things even worse. The second setup (also shown in Fig. 6), instead, scales very well also in case of violation. In this case, a CLP(FD) solver, written in CHR, directly manages the two alternative values for variable Answer.

The difference between the two setups of SCIFF and protocol becomes apparent in the worst case (i.e., the case of violation) when the whole tree is explored. With the first setup, the choice points left open in case of fulfillment and the disjunctions in the head of the integrity constraint make the number of nodes in the proof tree explode even for small number of bidders. With the second setup, instead, the tree has only one branch, and is thus explored in a reasonable time when the number of bidders increases.

5 Conclusions

In this paper, we have shown the results of the qualitative and experimental analysis of the computational behaviour of SCIFF (an abductive proof procedure used for verifying the compliance of agent interaction to interaction protocols) and SOCS-SI (the GUI-equipped system used to interface SCIFF to multi-agent systems).

The tests on laboratory-sized protocols show that SCIFF and SOCS-SI, although being research prototypes, can handle a number of messages which makes them usable in real applications such as electronic commerce.

However, the tests on a real protocol (combinatorial auction) show that the time costs of SCIFF may explode if the branching factor of the proof tree is high. In some cases (such as the combinatorial auction scenario shown in this paper) it is possible to reduce the branching factor by rewriting some of the integrity constraints (namely, by replacing disjunctions with domain variables). If the breadth of the proof tree is small, then the performance of SCIFF scales reasonably well.

One further step in this analysis, which we intend to pursue, is to identify a set of techniques such as the mentioned above to decrease the breadth of the proof tree (possibly automatic, based on syntactic conditions on the original protocol).

Future works will be dedicated also to study complexity formally, possibly building on the work by Eiter, Gottlob and Leone, who have shown [EG95, EGL97] that the complexity of the abduction model for the consistency problem is $\Sigma_2^P$.

Acknowledgments

This work has been supported by the European Commission within the SOCS project (IST-2001-32530), funded within the Global Computing Programme and by the MIUR COFIN 2003 projects La Gestione e la negoziazione automatica dei diritti sulle opere dell’ingegno digitali: aspetti giuridici e informatici and Sviluppo e verifica di sistemi multiagente basati sulla logica.

REFERENCES

of Applied Artificial Intelligence, Taylor & Francis, 2005.


